The Dissertation Committee for Jae Hong Min Certifies that this is the approved version of the following dissertation:

Fused Floating-Point Arithmetic for Application Specific Processors

Committee:

Earl E. Swartzlander, Jr., Supervisor

Lizy K. John

Mircea D. Driga

Nur A. Touba

Michael J. Schulte

Fused Floating-Point Arithmetic for Application Specific Processors

by

Jae Hong Min, B.S.; B.E.; M.S.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2013

Dedication

I dedicate this dissertation to my wonderful family

Acknowledgements

Foremost, I would like to express by sincere gratitute to Dr. Earl Swartzlander Jr. for making my graduate career a thoughtful and rewarding journey. His mentorship has truly inspired me as an engineer. I am greatly honored to have him as my advisor.

I would also like to thank my committee members: Dr. Lizy K. John, Dr. Mircea D. Driga, Dr. Nur A. Touba, and Dr. Michael J. Schulte for all their valuable comments and suggestions over the years.

I also thank my friends Junyoung Park, Young Taek Kim, Jae-Yong Jung, and Jong Ho Park for their friendship and invaluable feedbacks on my work which made my graduate school life joyful. I addition, I would like to thank my labmates: Sunghwan Kim, Jongwook Sohn, Kihwan Jun, and Bob Ascott for providing a delightful atmosphere in the lab.

Finally, I would like to thank my father Kyeongjin Min and my mother Jung Ae Jung for their encouragement, support, and their endless prayers during my whole graduate years. I also thank my sister, Song Ran Min for all the encouragement. And I thank my father in-law Chun-woo Yang, my mother in law Younghyo Yun, and my brother in law David S. Yang for their prayers and trust in me. And the person that I would like to thank most is my lovely wife Soo-Hyun who has been providing me with endless love, prayers, and happiness. Most of all, I would like to thank God for his immense love and guidance

Fused Floating-Point Arithmetic for Application Specific Processors

Jae Hong Min, Ph.D.

The University of Texas at Austin, 2013

Supervisor: Earl E. Swartzlander, Jr.

Floating-point computer arithmetic units are used for modern-day computers for 2D/3D graphic and scientific applications due to their wider dynamic range than a fixed-point number system with the same word-length. However, the floating-point arithmetic unit has larger area, power consumption, and latency than a fixed-point arithmetic unit. It has become a big issue in modern low-power processors due to their limited power and performance margins. Therefore, fused architectures have been developed to improve floating-point operations. This dissertation introduces new improved fused architectures for add-subtract, sum-of-squares, and magnitude operations for graphics, scientific, and signal processing.

A low-power dual-path fused floating-point add-subtract unit is introduced and compared with previous fused add-subtract units such as the single path and the high-speed dual-path fused add-subtract unit. The high-speed dual-path fused add-subtract unit has less latency compared with the single-path unit at a cost of large power consumption. To reduce the power consumption, an alternative dual-path architecture is applied to the fused add-subtract unit. The significand addition, subtraction and round units are performed after the far/close path. The power consumption of the proposed design is

lower than the high-speed dual-path fused add-subtract unit at a cost in latency; however, the proposed fused unit is faster than the single-path fused unit.

High-performance and low-power floating-point fused architectures for a two-term sum-of-squares computation are introduced and compared with discrete units. The fused architectures include pre/post-alignment, partial carry-sum width, and enhanced rounding. The fused floating-point sum-of-squares units with the post-alignment, 26 bit partial carry-sum width, and enhanced rounding system have less power-consumption, area, and latency compared with discrete parallel dot-product and sum-of-squares units. Hardware tradeoffs are presented between the fused designs in terms of power consumption, area, and latency. For example, the enhanced rounding processing reduces latency with a moderate cost of increased power consumption and area.

A new type of fused architecture for magnitude computation with less power consumption, area, and latency than conventional discrete floating-point units is proposed. Compared with the discrete parallel magnitude unit realized with conventional floating-point squarers, an adder, and a square-root unit, the fused floating-point magnitude unit has less area, latency, and power consumption. The new design includes new designs for enhanced exponent, compound add/round, and normalization units. In addition, a pipelined structure for the fused magnitude unit is shown.

Table of Contents

List of Tables

List of Figures

CHAPTER 1: INTRODUCTION

MOTIVATION AND GOAL

A floating-point number system represents a wider dynamic range than a fixed-point number system with the same word-length. The wide dynamic range is attractive for many digital signal processing (DSP) applications with the freedom from overflow/underflow and scaling concerns. However, the floating-point arithmetic unit has larger area, power consumption, and latency than a fixed-point arithmetic unit. It has become a big issue to design low-power floating-point arithmetic units to enable the floating-point units to be applied to mobile processors. Many techniques and architectures have been introduced for enhanced floating-point operations. To enhance floating-point operations, fused floating-point arithmetic units such as a fused multiply-add unit (FMA) [1,2], a single-path and dual-path fused add-subtract unit (fused A-S) [3,4,5], and a fused dot-product unit (fused DP) [6,7] have been developed.

The goal of this dissertation is to introduce new types of the fused floating-point architecture for sum-of-squares computation and magnitude operation to obtain enhanced performance, power, and area. In addition, this dissertation focuses on improving the dual-path fused add-subtract unit to reduce power consumption.

FLOATING-POINT NUMBERING SYSTEM

Floating-point computer arithmetic units are used for modern-day computers for 2D/3D graphic and scientific applications. The floating-point arithmetic unit can be implemented as a part of the general purpose CPU core, as an external co-processor, or as an application specific processor [8,9,10]. The floating-point arithmetic numbering system is defined in IEEE-754 standard [11].

Floating-Point Formats

The floating-point numbers consist of a sign, an exponent, and a significand. The one-bit sign represents the sign of the floating-point number (i.e., 0 for a positive number and 1 for a negative number). The size of the exponent mostly affects the dynamic range of the floating-point number. The significand consists of hidden one-bit integer and fractional bits. Equation (1.1) represents the floating-point number described by the sign, exponent, and significand.

$$N= (-1)^{sign} \times 2^{(exponent-bias)} \times significand \quad (1.1)$$

Various floating-point formats have been used such as single-precision, single-extended precision, double precision, and double-extended precision. Table 1.1 shows the precision formats [11]. This dissertation uses the single-precision format that is defined in IEEE-754 standard.

Table 1.1: Various Floating-Point Formats [11]

Floating-point format	Bit-width			
	Total	Sign	Exponent	Significand
Single (SP)	32	1	8	23
Single-extended	≧43	1	≧11	≧31
Double (DP)	64	1	11	52
Double-extended (EP)	≧79	1	≧15	≧63

Range of Floating-Point

The exponent is biased as shown in Equation (1.1). With the biased exponent, small numbers can be described by the floating-point number. For example, if the exponent is biased by 127 as described in the IEEE-754 single-precision standard, the positive maximum value (N_{max}) and minimum value (N_{min}) that the floating-point number represents are given by the following equations:

$$N_{max} = (-1)^0 \times 2^{(254-127)} \times 1.111\ldots.111_{(2)} \approx 3.4029 \cdots\cdots \times 10^{38} \quad (1.2)$$

$$N_{min} = (-1)^0 \times 2^{(1-127)} \times 1.000\ldots.000_{(2)} \approx 1.1755 \cdots\cdots \times 10^{-38} \quad (1.3)$$

The dynamic range is the ratio of the positive largest number (N_{max}) to the positive non-zero smallest number (N_{min}) that the floating-point represents. From N_{max} and N_{min}, the dynamic range is given by Equation (1.4).

$$\text{Dynamic Range} = N_{max} / N_{min} = (2^{253} \times 1.111\ldots111_{(2)}) \approx 2.8948 \times 10^{76} \quad (1.4)$$

The precision of the floating-point number is defined by the bit-width of the significand. Therefore, if the width bit is fixed, the precision and the dynamic range have a trade-off relationship.

When the exponent is small, the distance between adjacent floating-point numbers is very close (dense). On the other hand, with the large exponent, the distance between adjacent floating-point numbers becomes further (sparse). Figure 1.1 shows the range of a floating-point number [16].

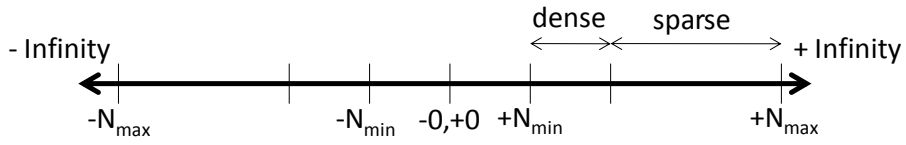


Figure 1.1: Range of a Floating-Point Number [16]

Special Floating-Point Numbers

There are special floating-point numbers such as zero, denormal, infinity, SNaN, and QNaN. For these special numbers, exception flags can be used. The floating-point zero can be expressed when the exponent and the significand are 0 regardless of the hidden bit. The sign bit can be used when the positive zero and negative zero need to be distinguished. When the exponent is zero and the significand is not zero, the floating-point number represents a denormal value. Denormal numbers can express values between $-N_{min}$ and N_{min}. Denormal numbers are equally spaced between $-N_{min}$ and N_{min}. Infinity is expressed by adding one to N_{max}. Table 1.2 shows the special floating-point numbers [11].

Table 1.2: Special Floating-Point Numbers [11]

Exponent	Significand	Result
00..0	00…0	Zero
00..0	0.00…1 to 0.11…1	Denormal (Subnormal)
00…01 to 11...10	1.00…00 to 1.11…1	Normal
11…1	1.00…0	Infinity
11…1	1.0…01 to 1.011..1	SNaN
11…1	1.1…01 to 1.11…1	QNaN

Rounding system

Floating-point arithmetic operations need a rounding process. The rounding process introduces round-off error. Four rounding modes such as Round-to-Zero (truncation), Round-to-Plus-Infinity, Round-to-Minus-Infinity, and Round-to-Nearest/Even (RNE) are frequently used [12]. For the Round-to-zero, bits below one ulp (unit in the last place) will be truncated. The truncated rounding process is the easiest and fastest with a larger round-off error compared to RNE. The Round-to-Plus-Infinity and Round-to-Minus-Infinity can be used for interval arithmetic with non-zero average round-off error.

For the Round to Nearest/Even (RNE), if the value below the ulp is larger than ½ ulp, rounding-up is performed. If the value below the ulp is exactly ½ ulp (halfway) and the rounded number will become even, rounding-up is performed as well. Table 1.3 shows the rounding-off error for RNE [16]. Figure 1.2 shows the four rounding modes with number lines [16].

Table 1.3: Rounding-up and Rounding-off Error of RNE (after [16])

upl	½ upl	¼ upl	Round-off Error
-	0	0	0
-	0	1	-0.25 upl
1→0	1	0	+0.5 upl
0→1	1	0	-0.5 upl
-	1	1	+0.25 upl

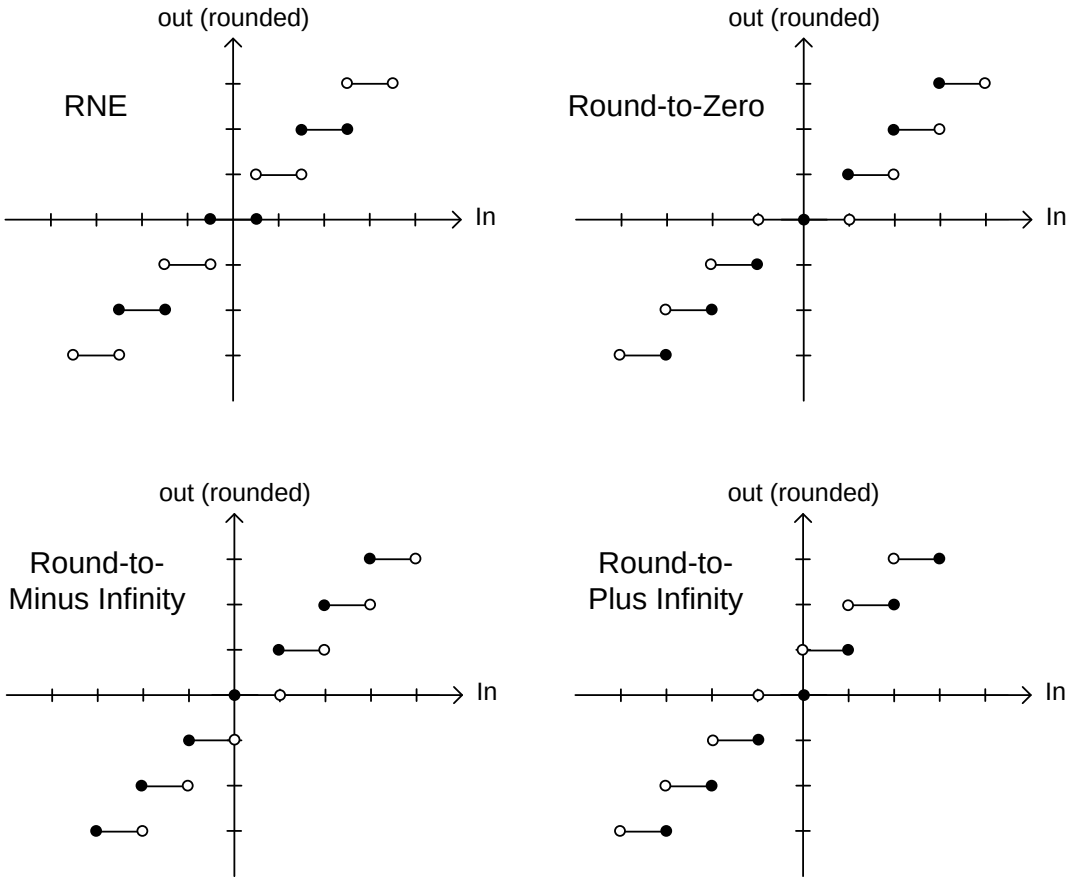
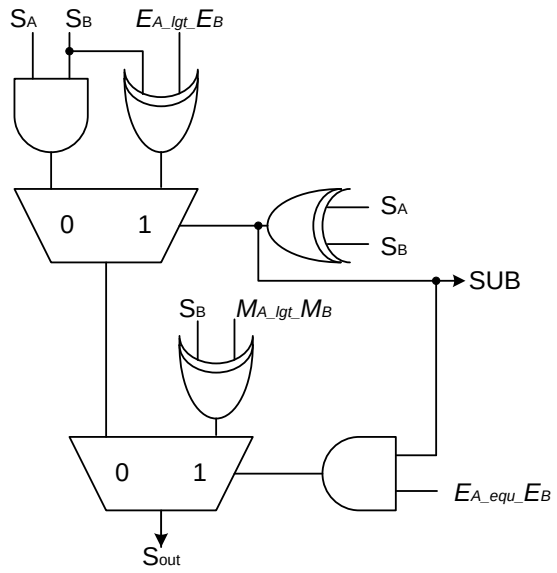


Figure 1.2: Rounding of Floating-Point Numbers (after [16])

FLOATING POINT ARITHMETIC UNIT – FLOATING-POINT ADDER

The two-input floating-point adder is one of the most frequently used floating-point arithmetic units. Many implementations have been proposed [13,14,15]. The floating-point adder consists of sign-bit decision unit, exponent processing unit, and significand processing unit. The sign-bit decision unit produces the true floating-point operation between addition and subtraction according to two sign bit inputs (S_A and S_B). If the two sign bits are the same, the floating-point adder performs addition. If not, the floating-point adder performs subtraction. The sign bit of the output is decided by the comparison of two exponents (E_A and E_B) and two significands (M_A and M_B). Figure 1.3 shows the sign decision logic. The signal $E_{A\_lgt\_}E_B$ is asserted when the exponent of input A is larger than the exponent of input B. If the significand of input A is larger than the significand of input B, the signal $M_{A\_lgt\_}M_B$ is asserted. The signal $E_{A\_equ\_}E_B$ is indicating that both of the exponents are the same.



S_{Are}	S_{Bre}	$S_{(Are+Bre)}$
0	0	0
0	1	EA>EB : 0 EA<EB : 1 EA==EB : (MA>MB : 0) or (MA<MB : 1)
1	0	EA>EB : 1 EA<EB : 0 EA==EB : (MA>MB : 1) or (MA<MB : 0)
1	1	1

Figure 1.3: Sign Decision Unit of a Floating-Point Adder

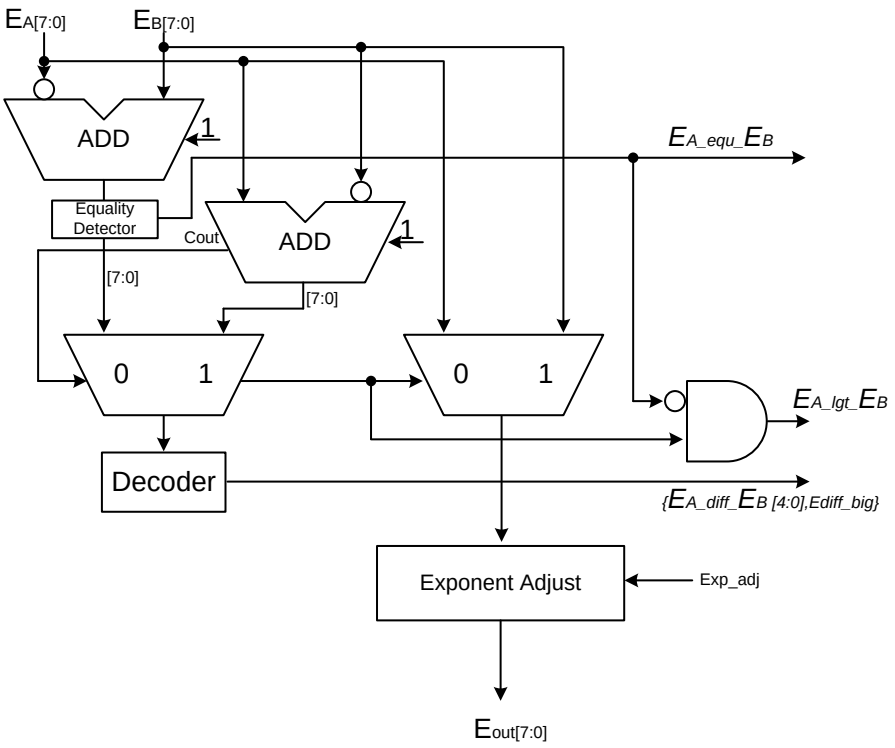


Figure 1.4: Exponent Processing Unit of a Floating-Point Adder (after [16])

The difference ($E_{A\_diff\_}E_B$) and comparison bits ($E_{A\_lgt\_}E_B$ and $E_{A\_equ\_}E_B$) of two input exponents are calculated in the exponent processing unit. Figure 1.4 shows the block diagram of the exponent processing unit [16]. If the exponent difference is larger than the bit-width of concatenation of the significand, guard, round, and sticky bit, the E_{diff_big} signal is asserted. When the E_{diff_big} signal is valid, only the 5 bit $E_{A\_diff\_}E_B$ signal is used for the significand alignment instead of the 8 bit difference, which reduces the size of the alignment unit.

The significand with the smaller exponent will be aligned. After the alignment, the aligned significand will be added to or subtracted from the significand with the larger

exponent. When the exponents are the same, the two significands are compared. In the case of subtraction, the smaller significand will be subtracted from the larger significand. After the significand addition/subtraction is executed, normalization, round, and post-normalization will be performed. The normalization step is required before the rounding step for higher-precision. Figure 1.5 shows the block diagram of the significand processing unit [17].

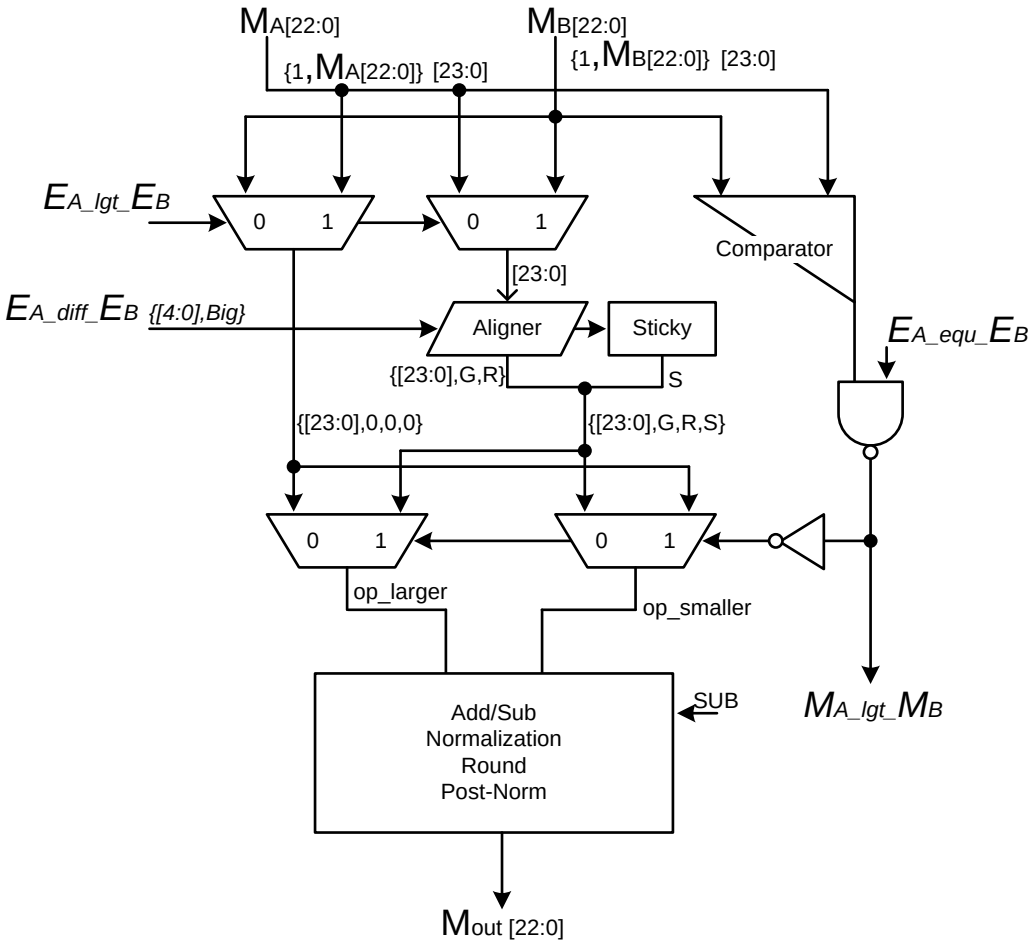


Figure 1.5: Significand Processing Unit of a Floating-Point Adder (after [17])

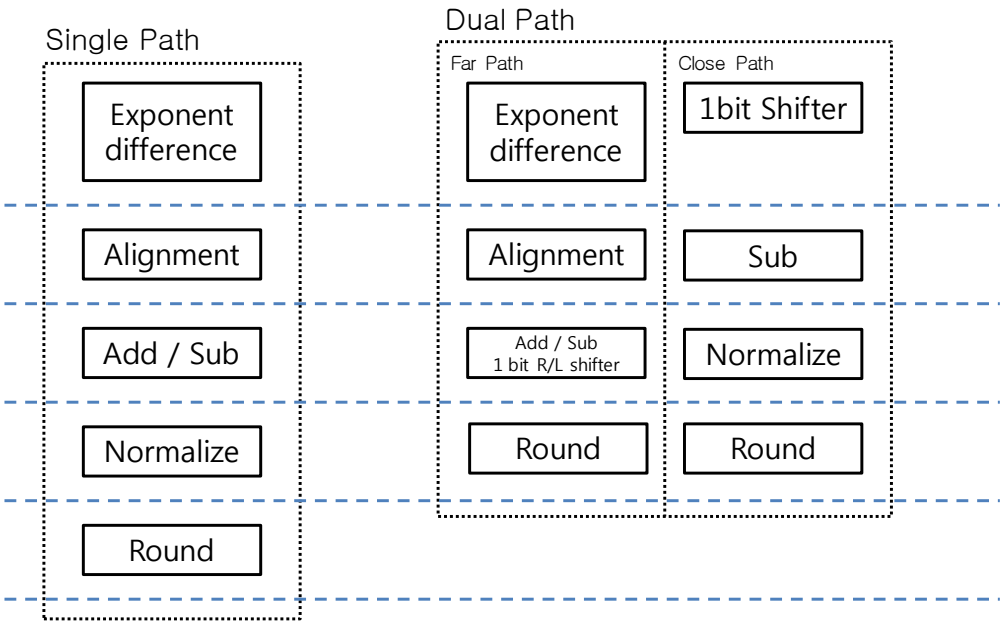


Figure 1.6: Dual-Path Floating-Point Adder Data Flow (after [16, 18])

For high-performance floating-point addition, Farmwald dual-path floating-point architecture is used [18]. When the floating-point adder executes true addition or subtraction with an exponent difference of more than one, the possible results are 1x.xxxxx…xxxx, 1.xxxxx…xxxx, and 0.1xxxx…xxxx. In these cases, a large general normalization is not required (only a one-bit shifter is needed). If the exponent difference is zero or one, the result may need general normalization for massive cancellation. However, a one-bit shifter can provide the alignment. Figure 1.6 represents the Farmwald dual-path floating-point data flow [16, 18].

$$1.1000\ldots.0010000 - 0.1000\ldots.0000100 = 1.0000\ldots.001100 \quad (1.5)$$

Massive cancellation needs a rounding process even if the rounding is not frequently computed. Equation (1.5) shows the case where the rounding process is necessary. The red digits indicate Guard, Round, and Sticky bit in order. The output of Equation (1.5) has non-zero guard bit. Therefore, rounding up may be required.

FLOATING POINT ARITHMETIC UNIT – FLOATING-POINT MULTIPLIER

A floating-point multiplier is also an important element of floating-point arithmetic. The floating-point multiplier has a simpler algorithm than the floating-point adder because complicated alignment and normalization is not necessary. For the sign bit of the output, one XOR gate is used as shown in Equation (1.6). In the exponent process of the floating-point multiplier, the two exponent inputs are added and subtracted by bias as shown in Equation (1.7).

$$S_{out} = S_A \; XOR \; S_B \tag{1.6}$$

$$E_{out} = E_A + E_B - \text{Bias} \tag{1.7}$$

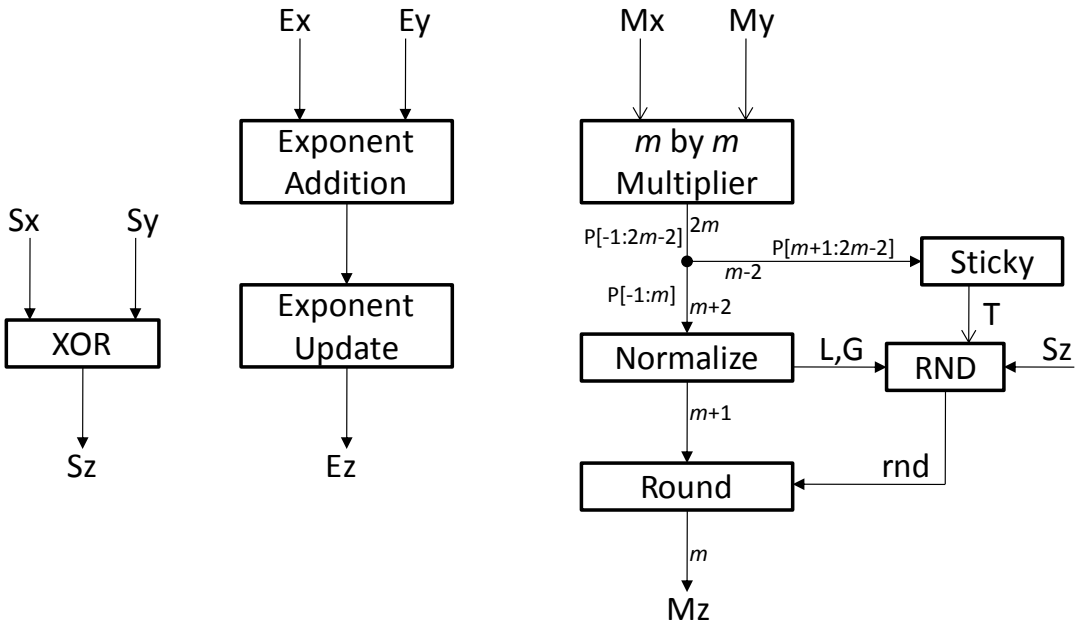


Figure 1.7: Block Diagram of a Floating-Point Multiplier [8, 19]

Since the range of the two significands are *[1, 2)*, the result of the significand multiplication is in the range of *[1, 4)*. Therefore, the result of the significand multiplication is required to be normalized by 1 bit right shifter. Figure 1.7 shows the block diagram of the floating-point multiplier [8, 19].

To increase performance, many techniques are employed. The significand multiplier tree produces sum and carry word. A combinational logic (non-adder) that generates the carry bit and sticky bit from the bottom half carry and sum word is faster and smaller than a logic that produces the carry bit as an output of the adder and the sticky bit from a big OR tree and the adder. The sticky generator indicates whether the sum of the bottom half words is zero or not. Figure 1.8 shows the sticky bit generator [8].

To generate the carry bit, a fast carry-lookahead chain is used. Figure 1.9 represents the block diagram of the enhanced floating-point multiplier [8]. PC and PS represent the carry word and the sum word, respectively.

S	s	s	s	...	s	s	s	s
C	c	c	c	...	c	c	c	c
-1	1	1	1	...	1	1	1	1
	z	z	z	...	z	z	z	z
	t	t	t	...	t	t	t	

$z_i = not\ (s_i\ xor\ c_i)$ $t_i = s_{i+1}\ or\ c_{i+1}$
$w_i = z_i\ xor\ t_i$
T(sticky bit)= *nand* (w_i)

Figure 1.8: Sticky Bit Generator (after [8])

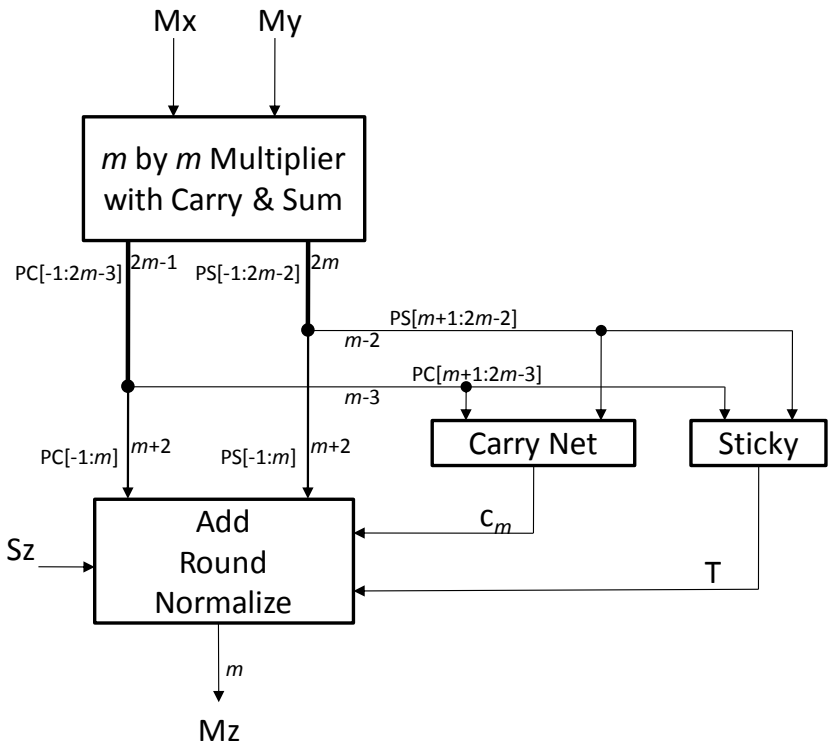


Figure 1.9: Block Diagram of the Enhanced Floating-Point Multiplier [8]

If the addition, round, and normalization are combined the performance will be increased. In this case, the rounding process is executed before normalization. Table 1.4 shows the two scenarios for the combined processing (Compound Add-Rnd-Norm) [8]. For scenario 1, the round bit is added in position *m*. On the other hand, the round bit is added in position *(m+1)* for scenario 2. The valid result will be selected from the two scenarios after completing the addition of the top half sum and carry word. Figure 1.10 represents the block diagram of the compound add-rnd-norm processing for the floating-point multiplier [8]. The intermediate result P1 is $PC+PS+(c_m+2)*2^{-m}$ for scenario 2 and P0 is $PC+PS+(c_m+1)*2^{-m}$ for scenario 1. P is selected from shifted P1 and P0 based on

P0[-1]. LADJ (Last Digit Adjustment Network) is required because one has been added in position *m* through the full adders of the first step. The L bit is P[*m*-1](P[*m*]+T*) for Round-to-Nearest-Even (RNE). The updated sticky bit T* is T+P1[*m*]*P0[-1].

Table 1.4: Scenarios for Compound Add-Rnd-Norm (after [8])

Scenario 1 : *rounding is requested and integer of top half addition is 1*
(-1) 0.123…(m-2)(m-1) m 0 1.xxx… x x x (PC+PS) c_m + 1
Scenario 2 : *rounding is requested and integer of top half addition is 2 or 3*
(-1) 0.123…(m-2)(m-1) m 1 x.xxx… x x x (PC+PS) c_m + 1

The algorithm as shown in Figure 1.10 requires addition of (c_m + 1) or (c_m + 2) based on the overflow condition. In other words, the range of the possible value in position *m* is *[1, 5]*. For example, the addition of position *m* is PS[*m*]+PC[*m*]+ (c_m + 1) or (c_m + 2). The addition in position *m* generates the *[0, 2]* range of the c_{m-1} value. If the range of c_{m-1} is *[0, 1]*, the valid result can be selected from the two c_{m-1} scenarios and the carry (c_m) can be removed from the critical path. To make the range of c_{m-1} as *[0, 1]*, one should be added in position *m*-1 (pre-addition). Table 1.5 shows the pre-addition cases and the ranges of c_{m-1} [8, 20]. Figure 1.11 shows the block diagram of the single-precision floating-point multiplier with the carry bit out of the critical path.

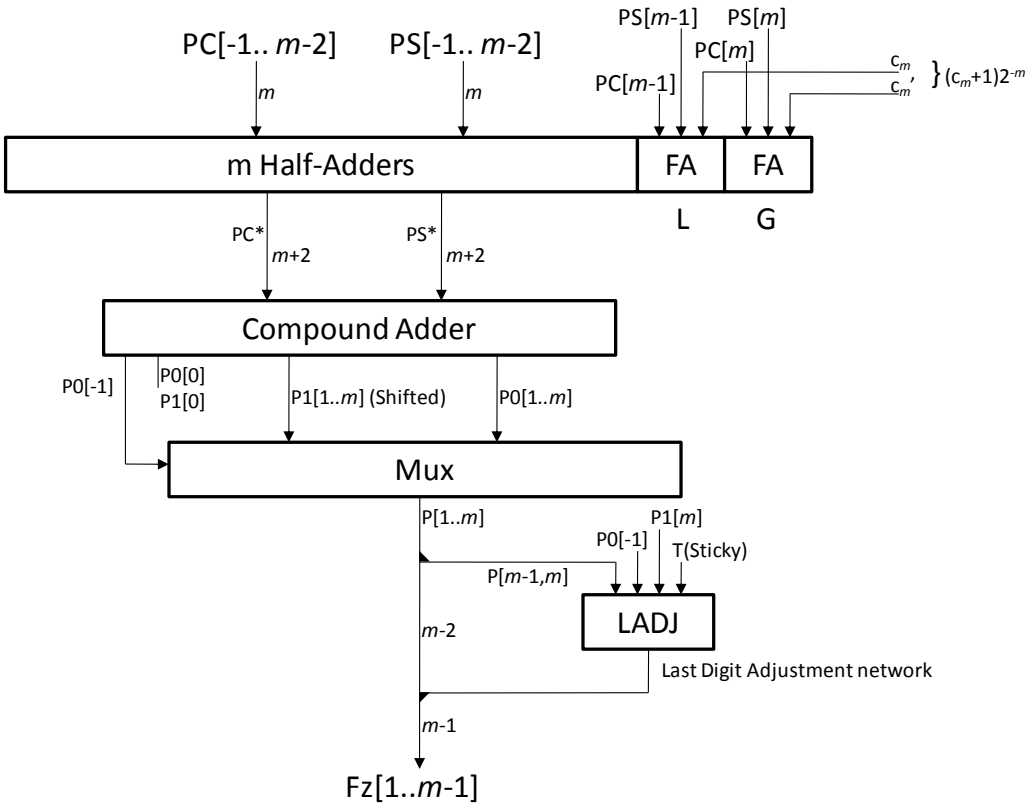


Figure 1.10: Block Diagram of Compound Add-Rnd-Norm [8]

Table 1.5: Pre-addition Cases and Range of C_{m-1} (after [8], [20])

PS[*m*]+PC[*m*] in position *m*	Range of ∑ before pre-add	Range of c_{m-1} without pre-add process	Pre-add	Range of ∑ after pre-add	Range of c_{m-1} with pre-add process
0	*[1,3]*	*[0,1]*	No	*[1,3]*	*[0,1]*
1	*[2,4]*	*[1,2]*	Yes	*[0,2]*	*[0,1]*
2	*[3,5]*	*[1,2]*	Yes	*[1,3]*	*[0,1]*

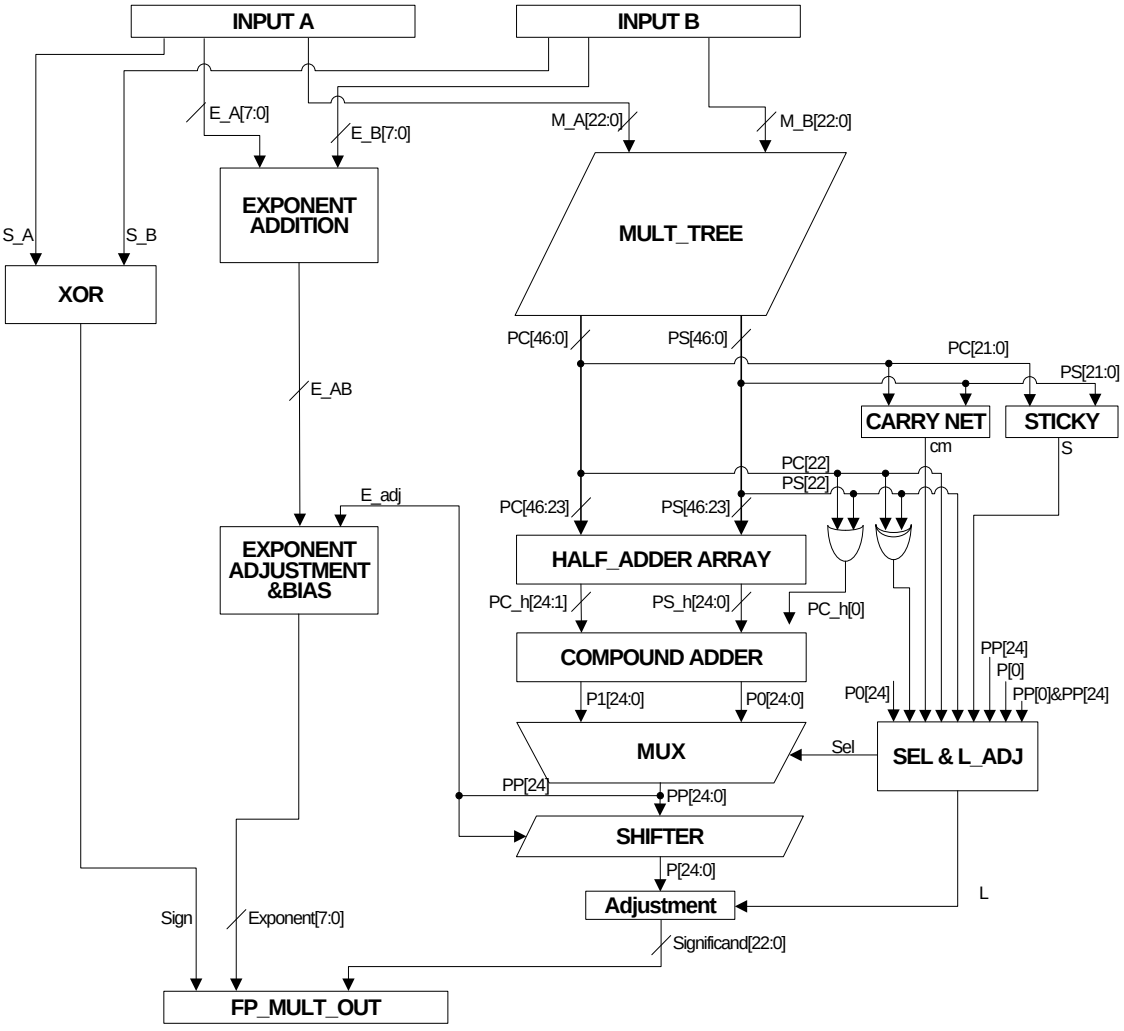


Figure 1.11: Block Diagram of Single-Precision Floating-Point Multiplier with Carry out of the Critical Path

CHAPTER 2: DESIGN METHODOLOGY

HIGH-LEVEL MODELING

Modeling by Python

The Python script language is used to build high-level floating-point arithmetic operation models [21]. The high-level models are used for functional verification and precision estimation. The high-level models are executed in sequence while the RTL models are executed in parallel. However, both Python and RTL models should be functionally equivalent. Algorithms 2.1 and 2.2 show pseudo-codes for the floating-point adder and multiplier/squarer.

The SIGN function decides the sign bit of the output. The MAX function compares two exponents and returns the larger exponent value. The STICKY function generates the sticky bit based on the amount of right-shift. The MASK function makes LSB = LSB *OR* Sticky bit. The MULT_TREE function provides a carry and sum word. However, the MULT_TREE function is not necessary for the floating-point multiplier and squarer. The overloading function "*" can substitute for the MULT_TREE function. The MULT_TREE function is built because it is used for the precision models of the fused units. The pseudo-codes assume that the exception decision is processed in the background when the exponent value is updated by the normalization and round step.

Algorithm 2.1 Pseudo-code of the Floating-Point Adder

```
START:
        Ediff = Ex – Ey
        Emax = Max (Ex , Ey)
        Sdiff = Mx – My
        Op = Sx ^ Sy
        Sout = SIGN(Ediff, Sdiff, Sx, Sy)
ALIGNMENT & SWAP:
    if Ediff > 0 :
        M_large = Mx
        M_small = My >> Ediff
        M_small_sticky = STICKY(My, Ediff)
        M_small = MASK ( M_small, M_small_sticky)
    elif Ediff < 0:
        M_large = My
        M_small = Mx >> Ediff
        M_small_sticky = STICKY(Mx, Ediff)
        M_small = MASK ( M_small, M_small_sticky)
    elif Sdiff > 0:
        M_large = Mx
        M_small = My
    else:
        M_large = My
        M_small = Mx
SIGNIFICAND ADDITION:
        Mz = Significand_addition/subtraction (M_large, M_small, Op)
NORMALIZATION:
        if (Length(Mz) > 24):
            Mz_norm = Mz >> 1
            Enorm = Emax + 1
            Mz_norm_sticky = STICKY(My, 1)
            Mz_norm = MASK ( Mz_norm, Mz_norm_sticky)
        elif (Length(Mz) < 24):
            Mz_norm = Mz << (24 – Length(Mz))
            Enorm = Emax – (24 – Length(Mz))
        else:
            Mz_norm = Mz
ROUND:
        Round_bit = RND (Mz_norm)
        Mz_rnd = (Mz_norm >> 3)+ Round_bit
        if (Length(Mz_rnd) > 24):
            Mz_rnd = Mz_rnd >> 1
            Ernd = Enorm + 1
        else:
            Ernd = Enorm
PACK:
        Sign = Sout
        Exponent = Ernd
        Significand = Mz_rnd
```

Algorithm 2.2 Pseudo-code of the Floating-Point Multiplier / Squarer

```
START:
        ExpXY = Ex + Ey – B
        Signxy = Sx ^ Sy
MULTIPLIER TREE:
        Man_XY_carry , Man_XY_sum = MULT_TREE (Mx, My) or SQ_TREE(Mx, My)
GENERATE CARRY & STICKY:
         Bottom_22bit_addition = Man_XY_carry & (2**22–1) + Man_XY_sum& (2**22–1)
         if (Length(Bottom_22bit_addtion ) > 22):
              Carry = 1
         else:
              Carry = 0

         Sticky_bottom = | (Bottom_22bit_addtion & (2**22–1))
SIGNIFICAND ADDITION:
          Product = (((Man_XY_carry>>22)+ (Man_XY_sum>>22) + Carry )<<1)+Sticky_bottom
NORMALIZATION:
        if (Length (Product) > 26):
             Product_norm = (Product >> 1)
             Enorm = ExpXY + 1
             Product _norm_sticky = STICKY(Product, 1)
             Product _norm = MASK (Product _norm, Product _norm_sticky)
ROUND:
        Round_bit = RND (Product _norm)
        Product _rnd = (Product _norm >>2) + Round_bit
        if (Length(Product _rnd) > 24):
             Product _rnd = Product _rnd >> 1
             Ernd = Enorm + 1
        else:
             Ernd = Enorm
PACK:
        Sign = Signxy
        Exponent = Ernd
        Significand = Mz_rnd
```

DESIGN AND IMPLEMENTATION FLOW

An ASIC design and implementation flow that is frequently used in industry is employed in this dissertation. The register-transfer level (RTL) models of the discrete and fused floating-point unit are designed using Verilog HDL 2001. The RTL model is compiled and simulated by Synopsys VCS and DVE [22]. Random values are used for the input vectors in the testbench files. For fast simulation, the random vectors are constrained to avoid excessive exception cases as shown in Algorithm 2.3 and 2.4.

The designs are implemented for various target frequencies using the Synopsys Design Complier with the Ultra™ RTL synthesis solution [24]. A bulk 45 nm process library is employed for the implementation. The Design Compiler produces Verilog-type gate-level netlists. Synopsys Formality® is used for functional equivalence-checking between the RTL model and the gate-level netlist model. Cadence Encounter, an Automatic Place and Route tool (APR), is used to generate layout models with standard cells provided from the bulk 45 nm process library [23].

Interconnect RC parasitic components (IEEE Standard Parasitic Exchange Format – SPEF) and standard delay format (SDF) are extracted from the layout models. The SDF file is used for back-annotation. A new netlist is generated from the layout model. Equivalence-checking by Formality is required for the layout netlist and the RTL model. The static timing and power consumption analysis of the gate-level netlist is executed by Synopsys PrimeTime PX ® [25]. Figure 2.1 shows the design and implementation flow.

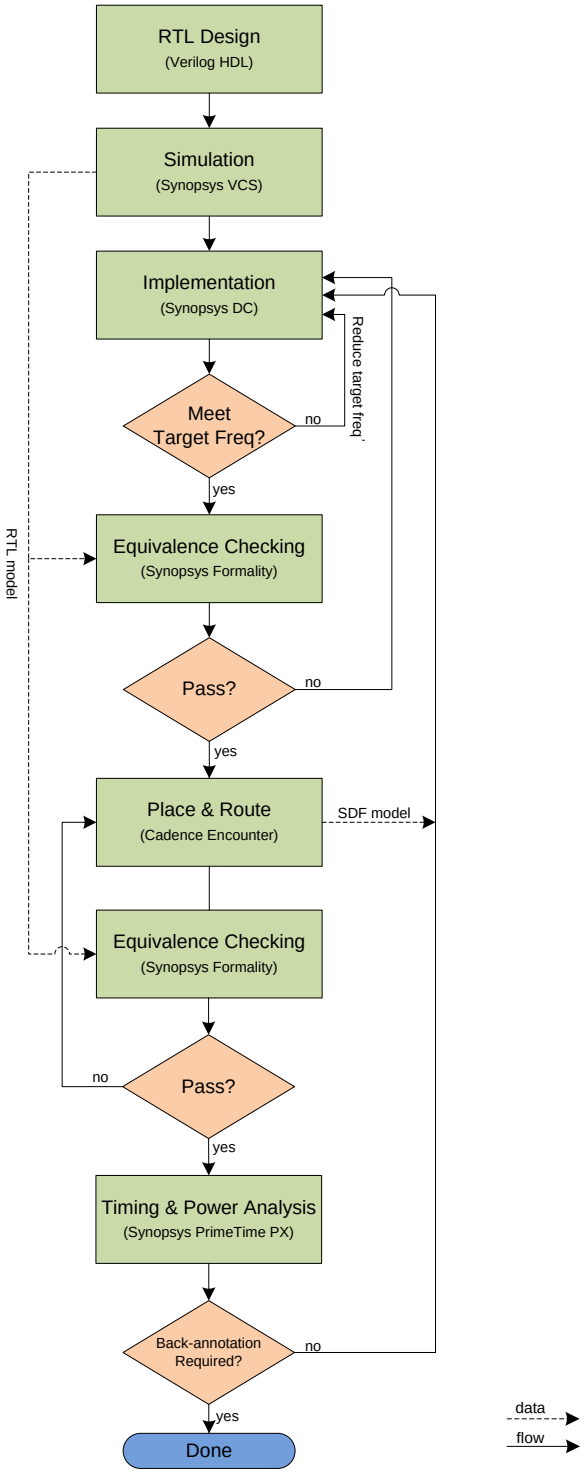


Figure 2.1: Design and Implementation Flow

Algorithm 2.3 Random Input Vector Generator for the Floating-Point Adder

```
parameter test_num = 1000000
initial begin
        for (i=0; i<1000000; i=i+1) begin
                #10
                IN_SA_rd = $random;
                IN_SB_rd = $random;
                IN_MA_rd = $random;
                IN_MB_rd = $random;
                IN_Exp = $random;
                ediff = $random;
                eselect = $random;

                if(eselect == 1) begin
                        IN_EA_rd = IN_Exp + ediff;
                        IN_EB_rd = IN_Exp; end
                else begin
                        IN_EA_rd = IN_Exp;
                        IN_EB_rd = IN_Exp+ediff; end

                while ((IN_EA_rd==9'h0FF)|(IN_EA_rd==9'h0)|(IN_EA_rd[8]==1)|(IN_EB_rd==9'hFF)|
              (IN_EB_rd==9'h0)|(IN_EB_rd[8]==1)) begin

                        IN_Exp = $random;
                        ediff = $random;
                        eselect= $random;

                    if(eselect == 1) begin
                                IN_EA_rd = IN_Exp + ediff;
                                IN_EB_rd = IN_Exp; end
                        else begin
                                IN_EA_rd = IN_Exp;
                                IN_EB_rd = IN_Exp+ediff; end
                end

                while ((IN_MA_rd==23'h7FFFFF)|(IN_MA_rd==23'h0)) begin
                        IN_MA_rd = $random; end

                while ((IN_MB_rd==23'h7FFFFF)|(IN_MB_rd==23'h0)) begin
                        IN_MB_rd = $random; end
        end
end
```

Algorithm 2.4 Random Input Vector Generator for the Floating-Point Multiplier

```
parameter test_num = 1000000
initial begin
        for (i=0; i<1000000; i=i+1) begin
                #10
                IN_SA_rd = $random;
                IN_SB_rd = $random;
                IN_MA_rd = $random;
                IN_MB_rd = $random;
                IN_EA_rd = $random;
                IN_EB_rd = $random;

                while ((IN_EA_rd==8'hFF)|(IN_EA_rd==8'h0)) begin
                        IN_EA_rd = $random; end

                while ((IN_EB_rd==8'hFF)|(IN_EB_rd==8'h0)) begin
                        IN_EB_rd = $random; end

                while ((IN_MA_rd==23'h7FFFFF)|(IN_MA_rd==23'h0)) begin
                        IN_MA_rd = $random; end

                while ((IN_MB_rd==23'h7FFFFF)|(IN_MB_rd==23'h0)) begin
                        IN_MB_rd = $random; end


                while    (((IN_EA_rd+IN_EB_rd-127) >= 255)|((IN_EA_rd+IN_EB_rd-127) <=0 )) begin
                        IN_EA_rd = $random;
                        IN_EB_rd = $random;

                        while (IN_EA_rd==8'hFF) begin
                                IN_EA_rd = $random; end
                        while (IN_EB_rd==8'hFF) begin
                                IN_EB_rd = $random; end
                end
        end
end
```

CHAPTER 3: LOW-POWER DUAL-PATH FLOATING-POINT FUSED ADD-SUBTRACT UNIT

OVERVIEW AND MOTIVATION

A floating-point add-subtract unit is used for DSP applications such as the fast Fourier Transform to reduce area and power consumption [26]. Figures 3.1 and 3.2 show the discrete and fused radix-2 fast Fourier Transform (FFT) unit with fused add-subtract units and fused dot-product units [27]. The fused FFT unit has 40% less power consumption, 35% less area, 15% less latency [27].

This chapter examines a low-power consumption dual-path fused floating-point add-subtract unit and compares it with previous fused add-subtract units such as the single path fused add-subtract unit and the high-speed dual-path fused add-subtract unit.

The high-speed dual-path fused add-subtract unit has less latency than the single-path fused add-subtract units, but with higher power consumption. To reduce the power consumption, a dual-path fused add-subtract unit with a simplified far/close path scheme is proposed; the significand addition, subtraction and round units are not included in the far/close paths while each path of the high-speed version has these units. The significand adder and subtractor are shared by far/close path. The power consumption of the proposed design is 18% lower than the high-speed dual-path fused add-subtract unit at a 16% cost in latency; however, the proposed dual-path fused add-subtract unit is 20% faster than the single-path fused add-subtract units.

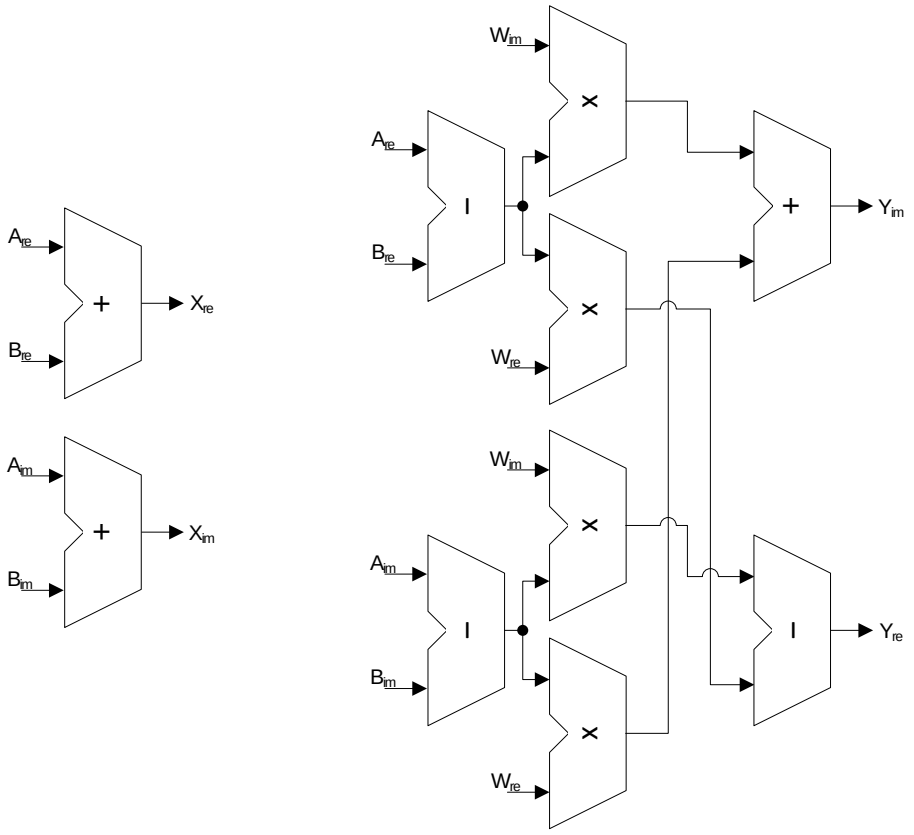


Figure 3.1: Discrete Radix-2 Butterfly Unit (after [27])

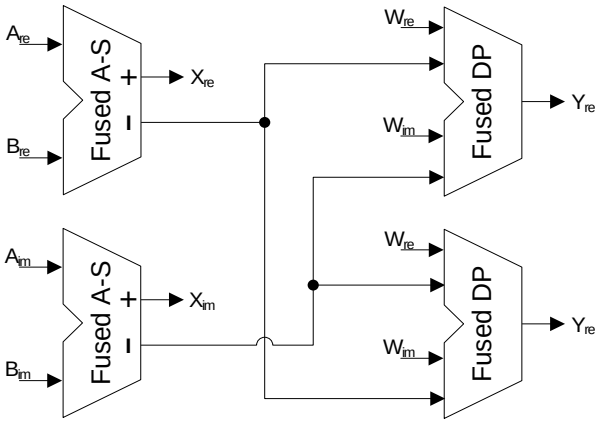
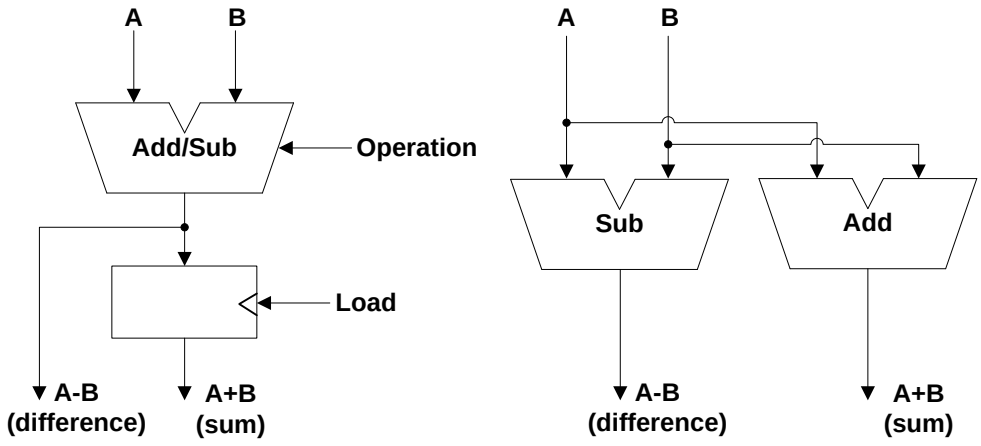


Figure 3.2: Fused Radix-2 Butterfly Unit (after [27])

DISCRETE ADD-SUBTRACT UNIT

The floating-point addition (sum) and subtraction (difference) can be executed with the same operands by a discrete serial add-subtract unit or a discrete parallel add-subtract unit. Figure 3.3 shows the discrete serial and parallel implementations. For the discrete serial add-subtract unit, one floating-point adder and a register are used [3]. An addition is performed in the first cycle and then a subtraction is performed in the second cycle; the result is small size, but large latency.

For the parallel discrete add-subtract unit, an addition and a subtraction are performed in parallel by two floating-point adders. The delay is about half as much as the serial discrete add-subtract unit, but the power consumption and area are about twice as much.



a) Serial AS b) Parallel AS

Figure 3.3: Discrete Serial and Parallel Add-Subtract Units [3]

SINGLE-PATH FUSED ADD-SUBTRACT UNIT

Naïve Single-Path Fused Add-Subtract

In the discrete add-subtract unit, the tasks of the exponent difference/compare unit and the significand swap/alignment unit are identical for the addition and the subtraction. Figure 3.4 shows the functionally identical logic between the two floating-point addition and subtraction units. In addition, normalization conditions between the addition and the subtraction are exclusive. The general normalization step is not required for addition.

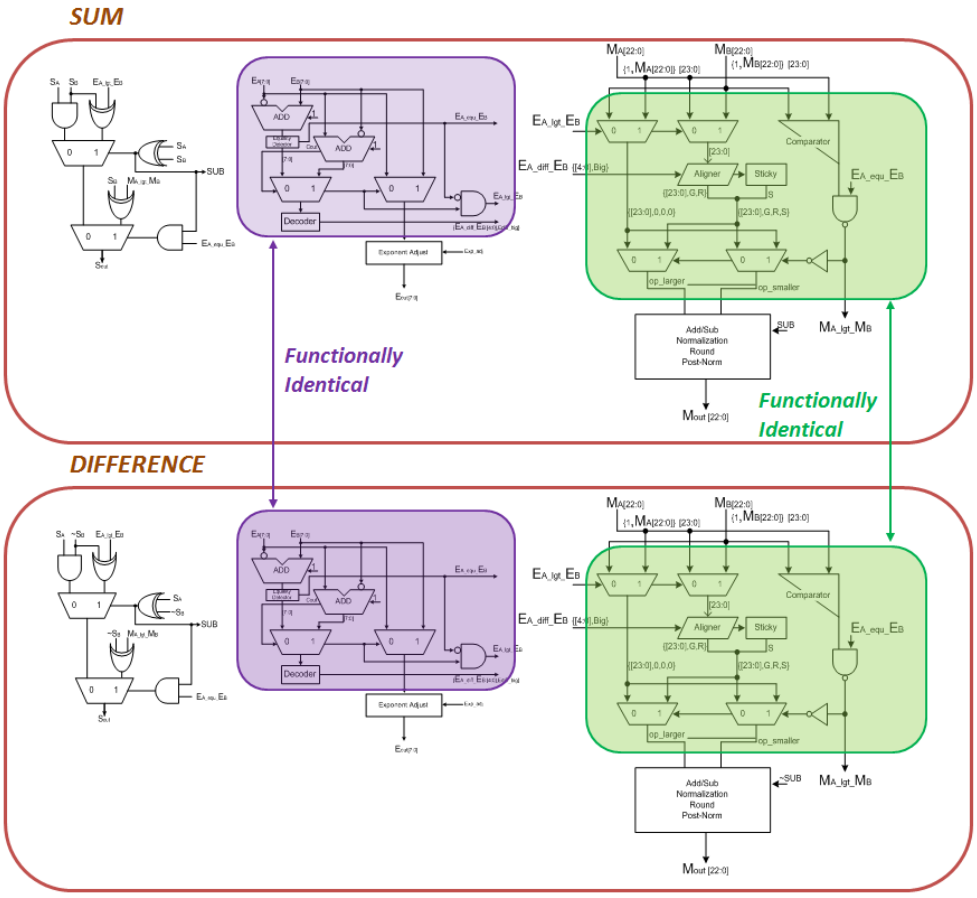


Figure 3.4: Identical Logic from the Discrete Add-Subtract Unit

Therefore, a part of the exponent unit and the significand processing can be shared by the addition and the subtraction. Furthermore, one of the two normalization steps can be removed. Figure 3.5 shows the fused add-subtract unit block diagram [3]. It has less area and power consumption compared to the discrete parallel implementation with a slight latency penalty. Table 3.1 presents a comparison between the three types of the add-subtract units [27]. The values in the table are relative to a single floating-point adder.

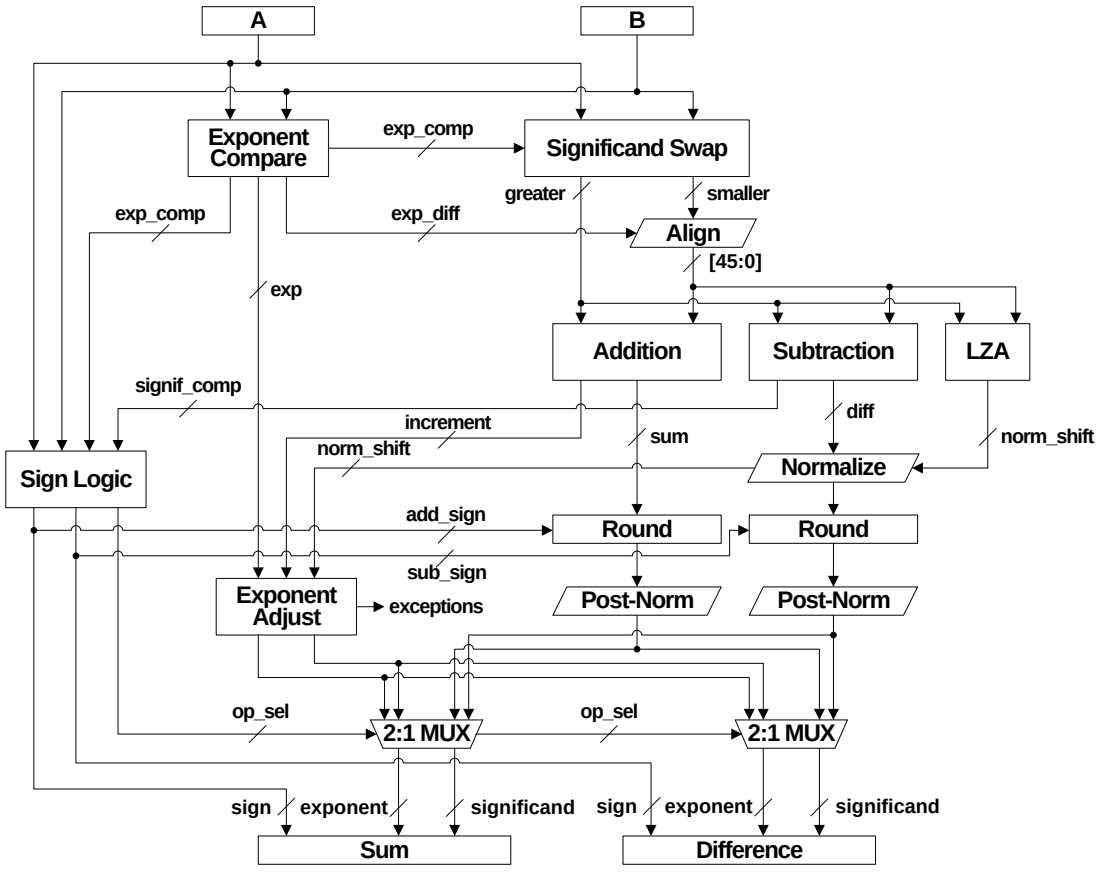


Figure 3.5: Floating-Point Fused Add-Subtract Unit (after [3])

Table 3.1: Comparison of Add-Subtract Implementations [27]

Add-Subtract Unit % of FPA	Serial	Parallel	Fused
Area	114%	196%	156%
Delay	208 %	104%	105%
Power	102%	190%	150%

As in the normal floating-point adder, the true floating-point operation is decided according to two sign bit inputs (S_A and S_B). For the floating-point subtraction (difference), the S_B is inverted. Figure 3.6 shows the block diagram of the sign decision unit.

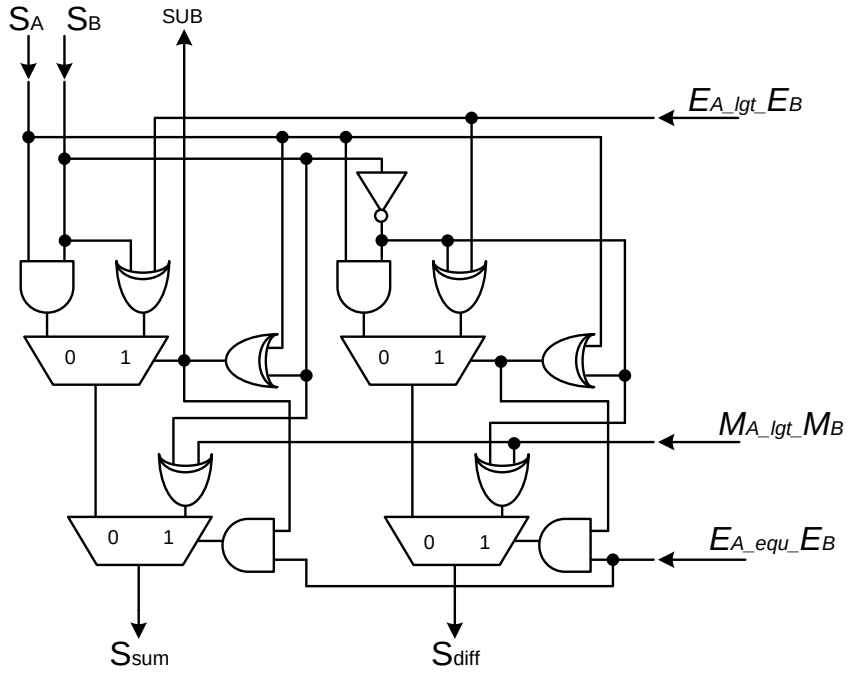


Figure 3.6: Sign Decision Unit of Floating-Point Add-Subtract

DUAL-PATH FUSED ADD-SUBTRACT UNIT

Dual-Path Algorithm

The dual-path algorithm is a common technique to reduce the latency of floating-point addition [14], [18]. The dual-path logic consists of a far path and a close path. The far path is a data path for addition and subtraction when the exponent difference is more than one. In the far path, a large normalization process is not necessary because the subtraction does not cause a massive cancellation. The close path is used when the exponents differ by 1, –1 or 0 where a large alignment is not necessary. For these three exponent difference cases, massive cancellation may occur after the significand subtraction. A leading zero anticipator (LZA) block works in parallel with the significand subtraction to detect zeros above the first one. Due to the simultaneous leading zero detecting, the close path latency is reduced.

High-Speed Dual-Path Fused Add-Subtract Unit

A dual-path fused add-subtract unit was proposed to reduce the latency [5]. Figure 3.7 shows the block diagram of the dual-path fused add-subtract unit. In the far path, adding, subtracting and rounding are executed in parallel as shown in Figure 3.8. The LZA is not necessary because massive cancellation does not occur for the far path. The two input significands are swapped based on the comparison of the exponents. The significand with the smaller exponent is aligned by the difference of exponents. Then the aligned significand and the unaligned significand are added, subtracted and rounded.

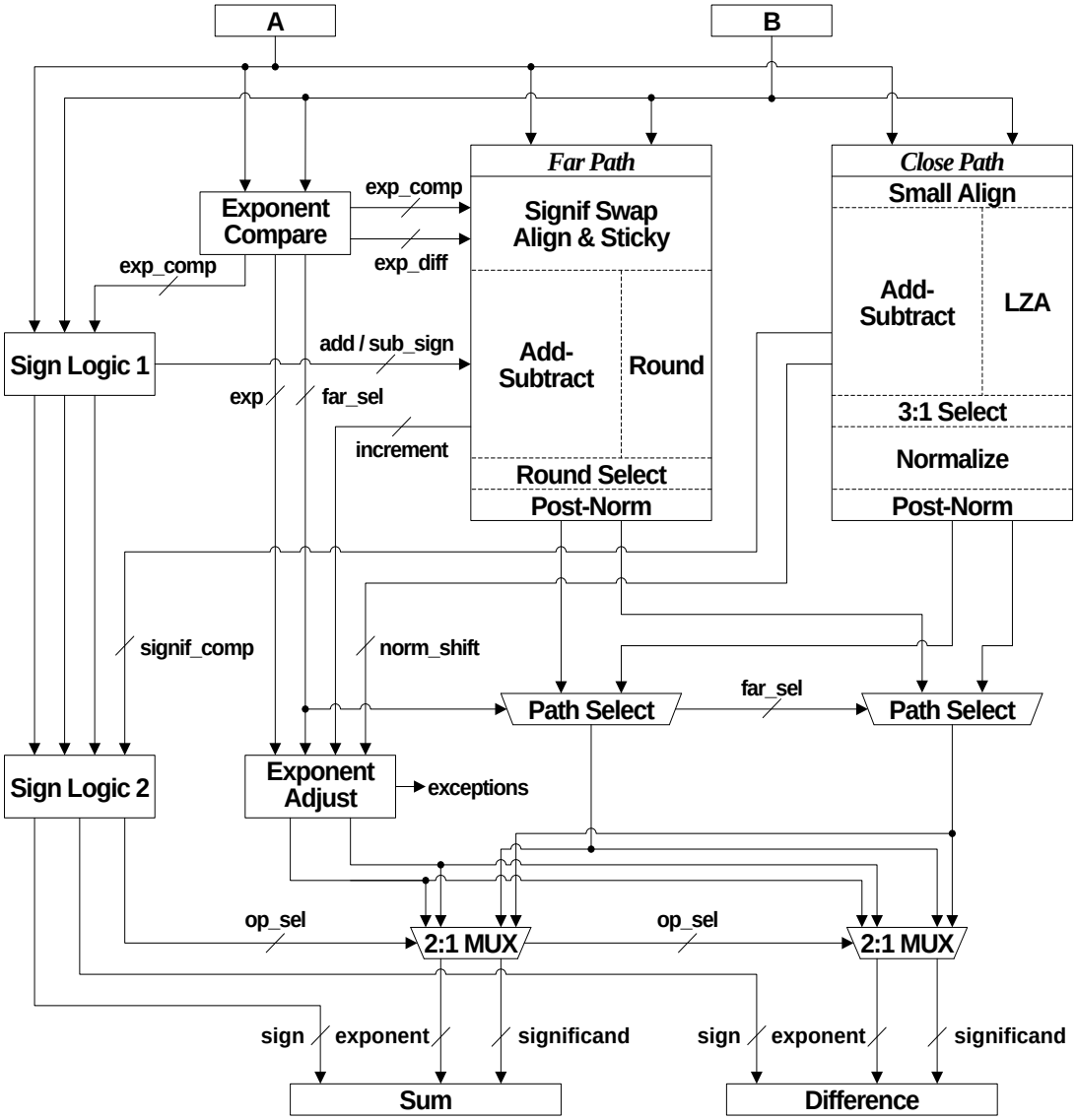


Figure 3.7: High-Speed Dual-Path Fused Add-Subtract Unit (after [5])

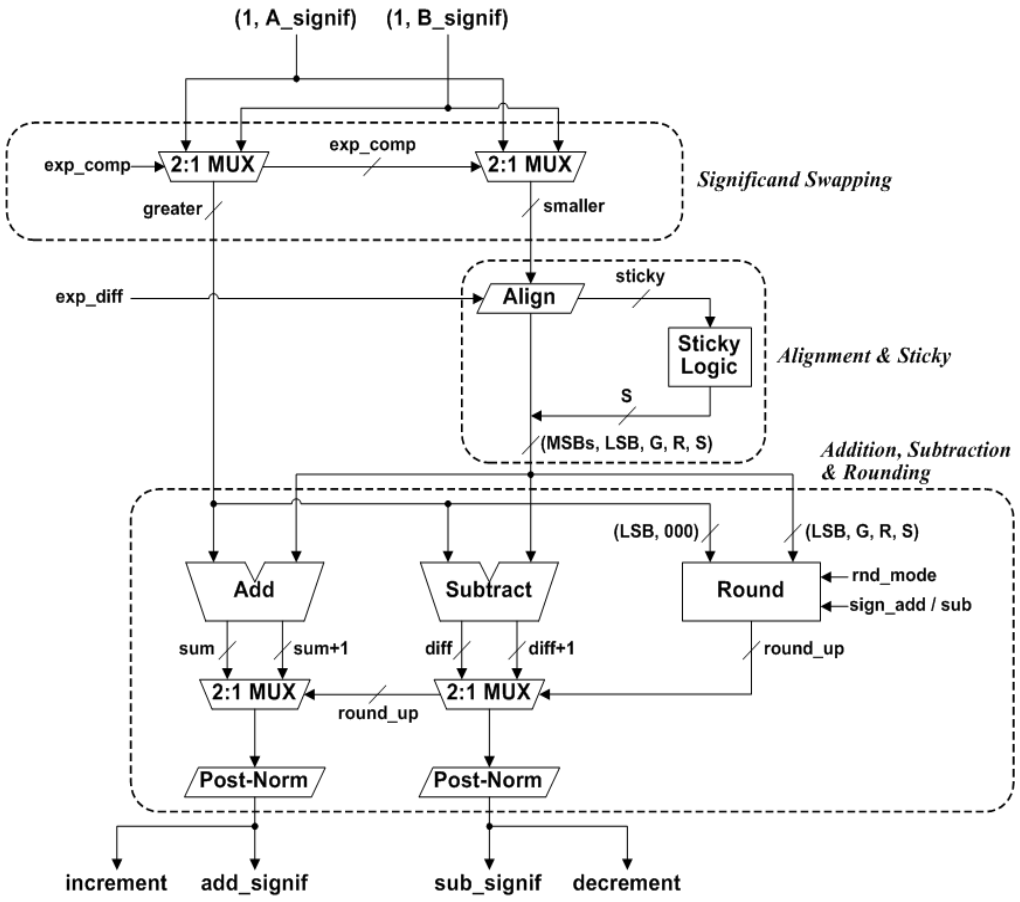


Figure 3.8: Far Path Logic of High-Speed Dual-Path Fused Add-Subtract Unit [5]

For the close path, three additions and subtractions and leading zero anticipations (LZAs) are performed in parallel for all three exponent difference cases {−1, 0, and 1} as shown in Figure 3.9. The latency is reduced with the parallel adders, subtractors and LZAs. Table 3.2 shows the parallel executions. After that, the valid result is selected and normalized. This parallel execution reduces the entire system latency because the close path is the critical path.

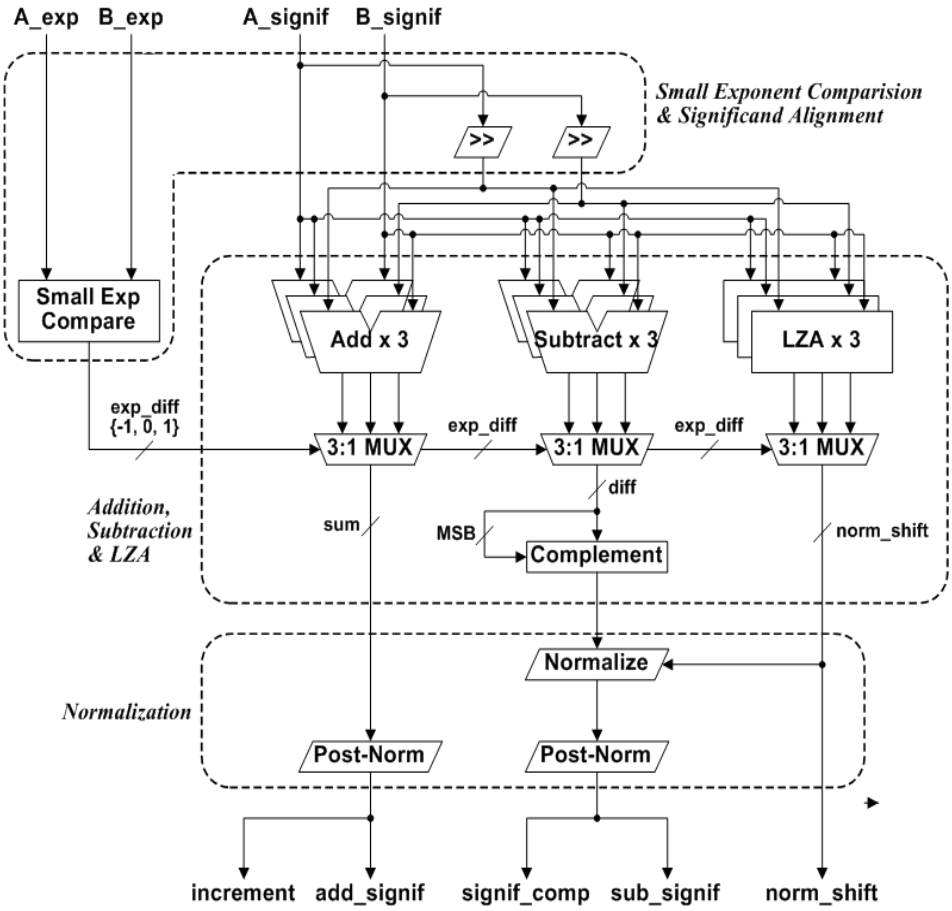


Figure 3.9: Close Path Logic of High-Speed Dual-Path Fused Add-Subtract Unit [5]

Table 3.2: Parallel Execution in Close Path

Add (Significand A, Significand B >>1) Add (Significand B, Significand A>>1) Add (Significand A, Significand B) Sub (Significand A, Significand B >>1) Sub (Significand B, Significand A>>1) Sub (Significand A, Significand B) *if negative, complemented LZA (Significand A, Significand B >>1) LZA(Significand B, Significand A>>1) LZA(Significand A, Significand B)

PROPOSED DUAL-PATH FUSED ADD-SUBTRACT UNIT

The parallel additions, subtractions and LZAs of the high-speed dual-path fused add-subtract unit reduce the latency, but at a cost of large power consumption. Table 3.3 shows the percentage of the power consumption of the parallel adders and subtractors in the far/close path (21.8% for close-path execution). The parallel additions and subtractions of the close path can be removed to reduce the power consumption in the proposed dual-path fused add-subtract unit.

Table 3.3: Percentage of Power Consumption of Add/Subtract/Round Units

Far Path		Close Path	
Adder	4.5 %	Adders	6.0 %
Subtractor	6.9 %	Subtractors / 2's Comp	10.4 %
Round / Select	1.9 %	Select	5.4 %

The alternative approach uses a different dual-path architecture that is based on a design for a floating-point adder [17]. This dual-path architecture can be modified and applied for a fused add-subtract unit. Figure 3.10 shows the block diagram of the proposed dual-path fused add-subtract unit. The dual-path logic for the proposed fused add-subtract unit is simpler than the dual-path for the high-speed dual-path fused approach described in the previous section because the far/close paths do not have adders and subtractors.

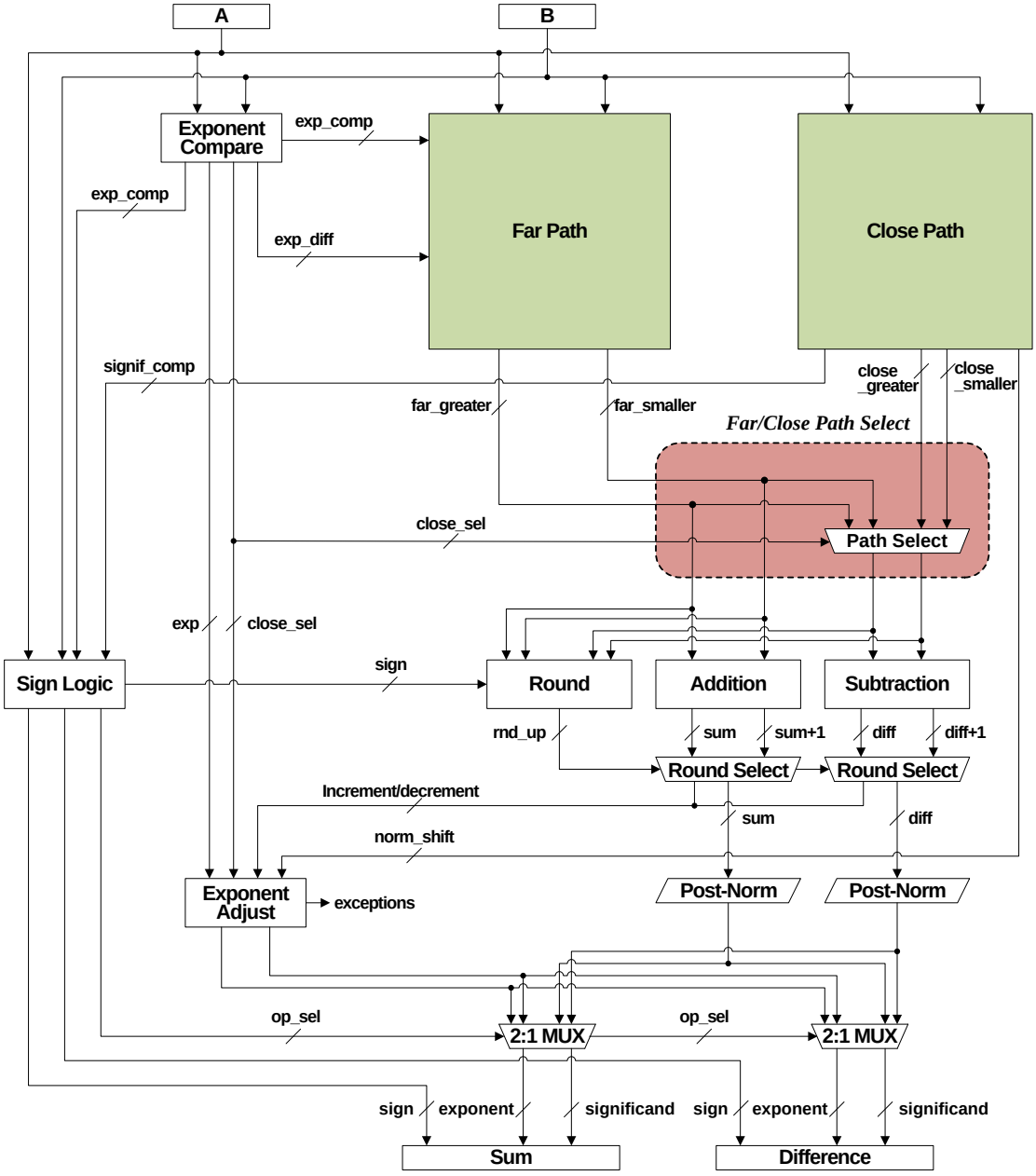


Figure 3.10: Proposed Dual-Path Fused Add-Subtract Unit

The addition and subtraction can be shared after the far and close path. This reduces the number of adders and subtractors from that of the high-speed dual-path fused add-subtract unit. However, the latency is increased because the addition and subtraction are performed after far/close path.

The significand inputs are pre-processed with the alignment and normalization of the far/close path. The far path has significand swapping logic based on the exponent comparison. The smaller significand is shifted by the exponent difference. In the close path, three LZAs and two normalization units are used. The LZAs are performed in parallel for all input cases such as (Significand A, Significand B >>1), (Significand B, Significand A>>1) and (Significand A, Significand B).

The aligned input and the three parallel LZAs reduce the delay. Among the three LZA outputs, the proper LZA output is selected for the normalization step by the exponent comparison. The normalization is performed before the subtraction to avoid massive cancellation. Figures 3.11 and 3.12 present the proposed close path and far path blocks. In the close path, the critical path is the LZA and the normalization. The close path block for the proposed fused add-subtract unit is smaller than the close path block for the high-speed dual-path unit because the adders and subtractors are not included in the close path. After the far/close path processing, the significand data is passed based on the conditions that are shown in Table 3.4.

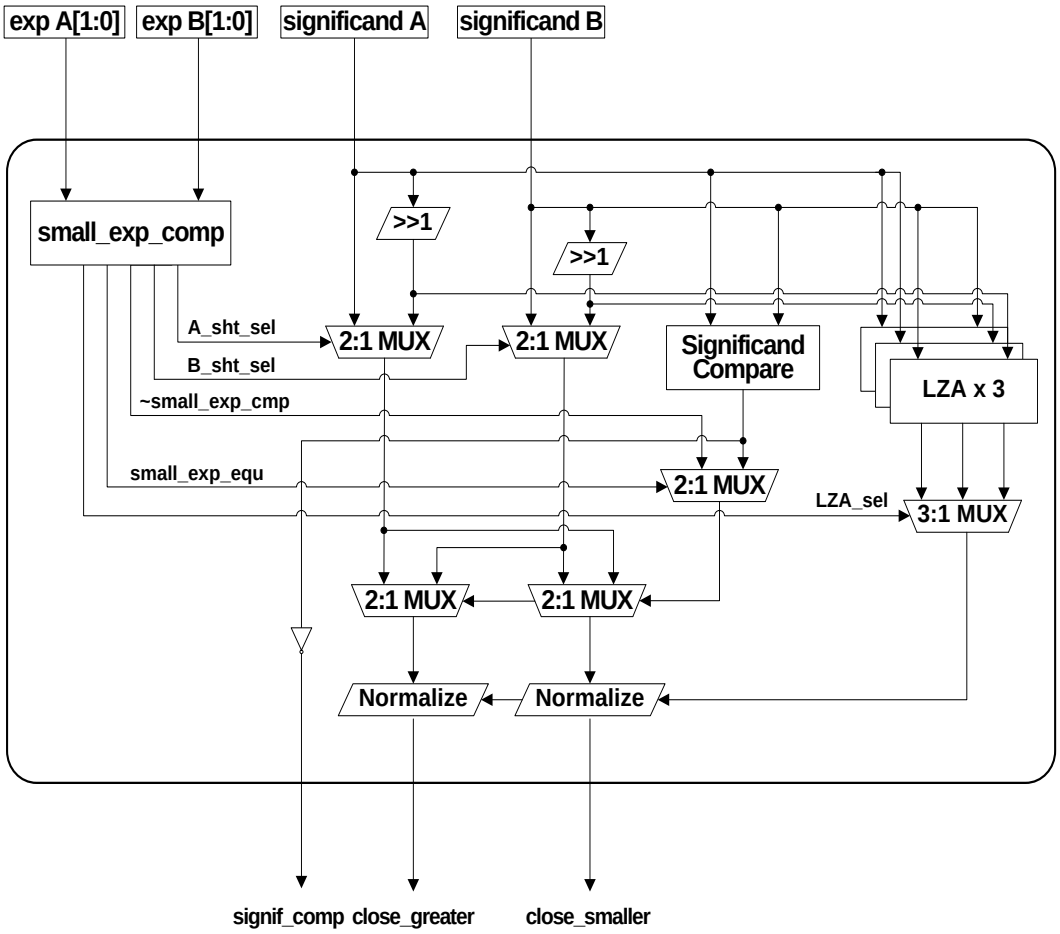


Figure 3.11: Close Path Block for the Proposed Dual-path Fused Add-Subtract Unit

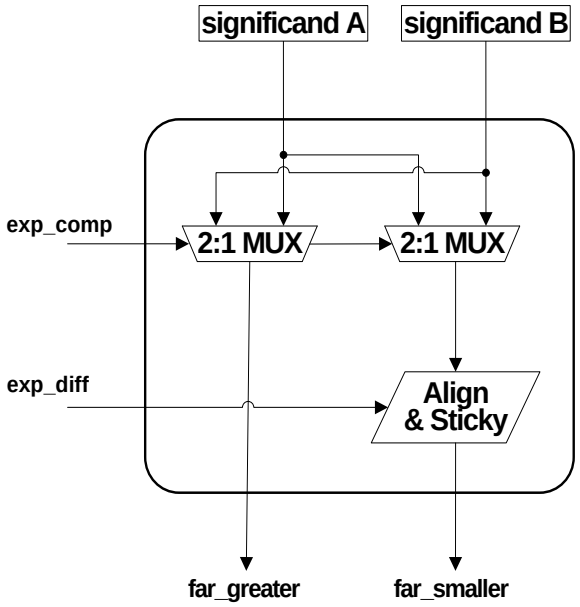


Figure 3.12: Far Path Block for the Proposed Dual-path Fused Add-Subtract Unit

Table 3.4: Path Select Table Between Far/Close Path and Adder/Subtractor

|exp difference|	Adder input	Subtractor input
0,1	*from* far path	*from* close path
>1		*from* far path

The naïve single-path fused add-subtract unit, the high-speed dual-path and the proposed low-power dual-path fused add-subtract unit have been implemented and compared. The functionality of the three types of fused add-subtract units were verified by MATLAB. Algorithm 3.1 shows the example MATLAB code to verify the close path block.

Algorithm 3.1 MATLAB Code to Verify the Close Path Logic

```
error_flag=0;
      while (k <10000000)
                  a_dec=(randi([4194304 16777215]));
                  b_dec=(randi([4194304 16777215]));
                  size(dec2bin(a_dec));
                  size(dec2bin(b_dec));
                  sub_dec = abs( a_dec - b_dec );
                  a= size(dec2bin(sub_dec));
                  size_out=a(2);

                  if (size_out <24)
                              shifted_out = bitsll(sub_dec, (24-size_out));
                  else
                              shifted_out = sub_dec;
                  end

                  shifted_a = bitsll(a_dec, (24-size_out));
                  shifted_b = bitsll(b_dec, (24-size_out));
                  low_power_out = abs(shifted_a - shifted_b);
                  k=k+1;

                  if (shifted_out ~= low_power_out)
                              error_flag =1;
                  end
            end
```

Then they were implemented and simulated by Synopsys and Cadence CAD tools as described in Chapter 2. Figure 3.13 examines the relative ratios of the area, power consumption, and latency of the three fused add-subtract units from the Design Compiler result with various synthesis options. The average for the latency of the Naïve single-path design is 130%. The average power consumption and latency of the high-speed dual-path design are 120% and 86% relative to the proposed dual-path add-sub unit.

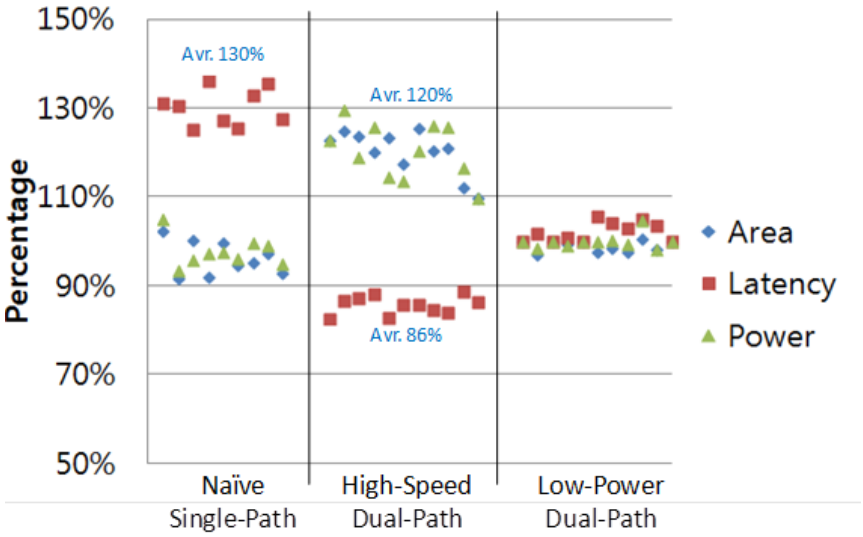


Figure 3.13: Comparison of the Various Fused Add-Subtract Units

Table 3.5: Comparison of the Three Types of Fused Add-Subtract Units

Path Type	Fused Add-Subtract Unit Type	Standard Cell Area (μm^2)	Total Power (mW)	Latency (ns)
Single Path	Naïve	3,063 (100%)	0.85 (100%)	2.34 (100%)
Dual Path	High-Speed	4,348 (142%)	1.22 (144%)	1.61 (69%)
	Low-Power	3,495 (114%)	1.00 (118%)	1.87 (80%)

Table 3.5 shows the cell area, power consumption (@ 300MHz) and latency for the three types of the fused add-subtract units. The latency of the dual-path fused add-subtract units are less than that of the single path fused add-subtract units at cost of increased power consumption and area. The power consumption of the proposed low-power dual-path fused add-subtract unit is 18% less than the high-speed dual-path fused add-subtract unit. The proposed fused add-subtract unit is 16% slower than the high-speed dual-path fused add-subtract unit. Figure 3.14 examines the relative ratios of the area, power consumption, and latency relative to the proposed dual-path fused add-subtract unit.

The latency of the proposed dual-path fused add-subtract unit is 20% less than the latency of the naïve single-path fused add-subtract unit. Table 3.6 shows the critical path of the proposed dual-path add-sub unit. Figure 3.15 shows the layout of the proposed fused add-subtract unit. The supply voltage is 1.1V. Metal layers 1 to 10 are used. The die size is 100um x 100um. The standard cell core density is 69.1%.

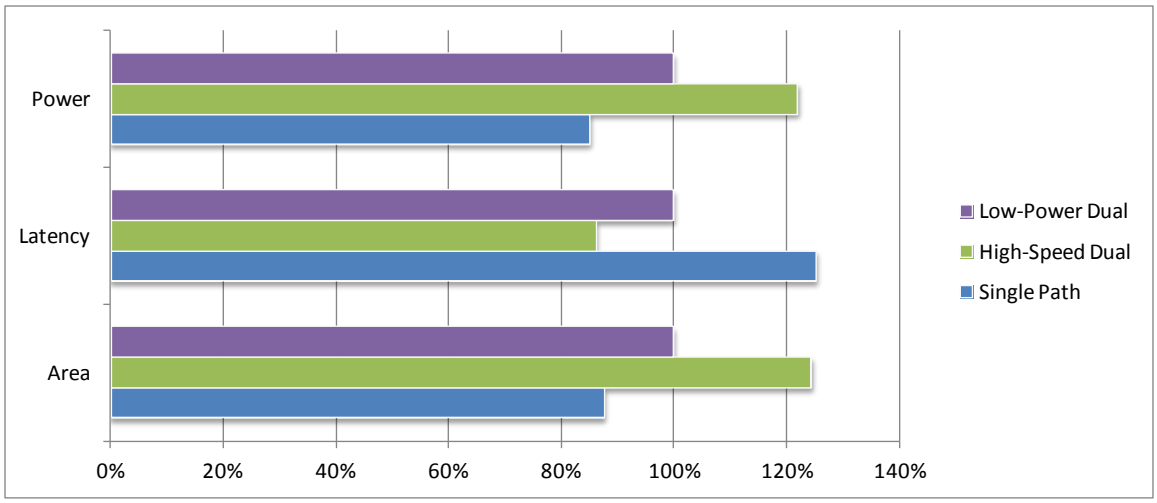


Figure 3.14: Relative Ratios of the Fused Add-Subtract Units

Table 3.6: Critical Path of the Proposed Low-Power Dual-Path Add-Subtract Unit

Critical Path		
Unit Type	Latency (ns)	Percentage
LZA	0.60	32%
LZA_sel	0.23	12%
Normalization	0.40	21%
Path_sel	0.05	3%
Sub/Rnd/Rnd_sel	0.39	21%
Post_norm/ Exception/Mux	0.20	11%
Total Latency	1.87	100%

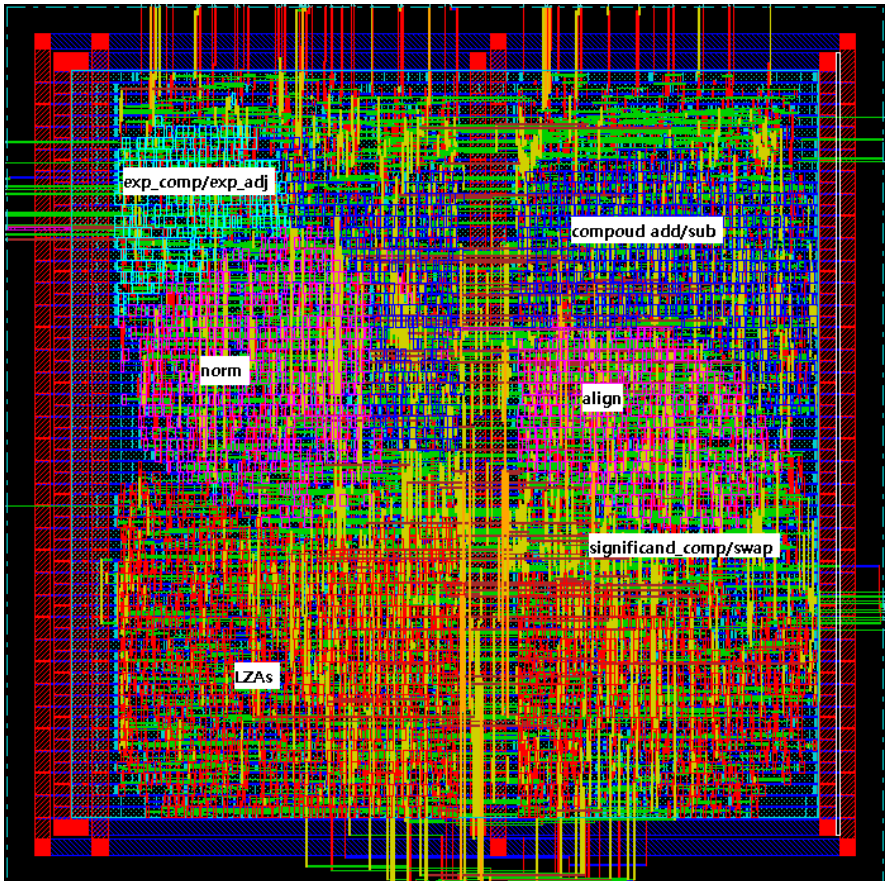


Figure 3.15: Layout of the Proposed Low-Power Dual-Path Fused Add-Subtract Unit

CHAPTER 4: FLOATING-POINT FUSED TWO-TERM SUM-OF-SQUARES UNIT

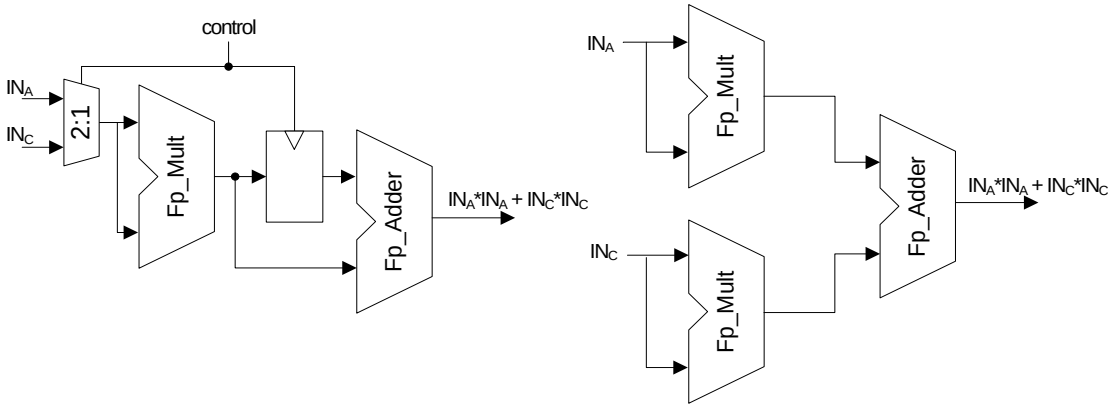
OVERVIEW AND MOTIVATION

A sum-of-squares operation is used for many applications such as filtering, Euclidian branching, pattern recognition, vector normalization, and a compound collision operation [29, 30]. The sum-of-squares is also used to compare the magnitude of complex numbers. Sum-of-squares computations can take two terms or multi-terms. This dissertation focuses on the two-term sum-of-squares computation with enhanced performance and low power consumption.

This chapter introduces new fused floating-point architectures for two-term sum-of-squares computation to obtain enhanced performance, power consumption, and area. The fused architectures include pre/post-alignment, partial carry-sum width, and normal/enhanced rounding. Hardware tradeoffs are presented between the fused designs in terms of power consumption, area, and latency. The low-power and high-performance fused floating-point two-term sum-of-squares (fused SoSQ) units are compared with discrete dot-product units containing normal significand multipliers and discrete sum-of-squares units containing significand squarers. The fused floating-point sum-of-squares unit with 26 bit partial carry-sum processing, post-alignment, and enhanced rounding has 38% less power consumption, 23% less area, and 49% less latency compared with discrete parallel sum-of-squares units containing a standard floating-point adder.

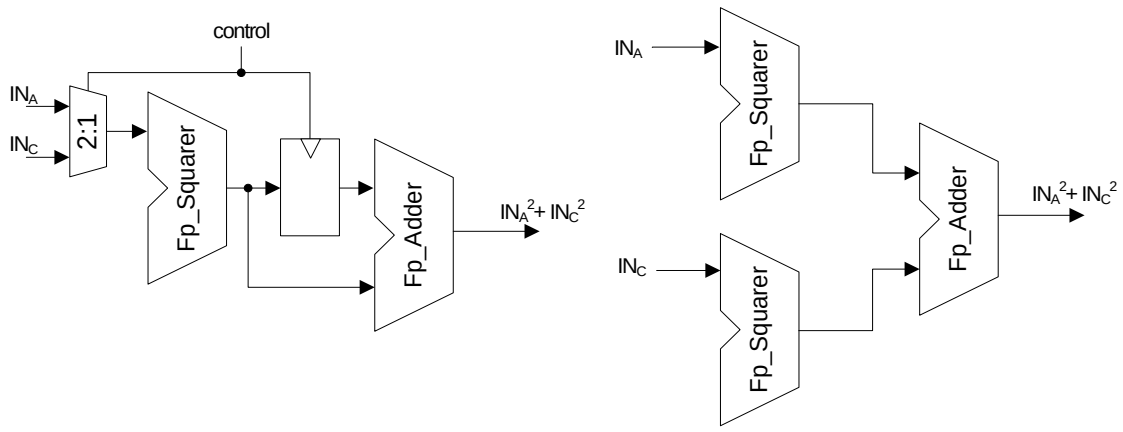
DISCRETE TWO-TERM SUM-OF-SQUARES UNIT

The floating-point sum-of-squares computations can be executed by general discrete floating-point architectures such as a discrete serial or parallel floating-point dot-product unit implemented with conventional floating-point multipliers and a floating-point adder. Figure 4.1 shows the discrete serial and parallel floating-point dot-product architectures. For the serial unit, a floating-point multiplier, a floating-point adder, a multiplexer, and a register are used. It needs two cycles to compute the sum-of-squares. The discrete floating-point parallel dot-product unit is implemented with a floating-point adder and two multipliers that are synthesized separately. The parallel floating-point dot-product units have larger area and power consumption, but much lower latency than the serial floating-point dot-product unit.



a) Serial dot-product b) Parallel dot-product

Figure 4.1: Discrete Floating-Point Dot Product Unit (after [6])



a) Serial Sum-of-Squares b) Parallel Sum-of-Squares

Figure 4.2: Discrete Floating-Point Sum-of-Squares Unit

The discrete floating-point sum-of-squares unit can also be implemented with conventional floating-point squarers and a floating-point adder. The floating-point adder has functions for addition and subtraction even though the sum-of-squares computation does not need the subtraction; this redundant function increases the area, power and latency. Figure 4.2 shows the discrete serial and parallel floating-point sum-of-squares architectures. The discrete sum-of-squares architecture is similar to the discrete dot-product architecture except that floating-point squarers are used in place of floating-point multipliers.

Floating-point Adder, Multiplier, and Squarer Designs

Floating-point adders, multipliers and squarers are designed and implemented for the sub-blocks of the discrete floating-point dot-product and sum-of-squares unit. A single-path adder is selected and implemented as described in Chapter 1.

A single-precision floating-point multiplier and squarer with Dadda reduction and advanced carry and sticky network are designed. The floating-point multiplier shown in Figure 1.11 in Chapter 1 is employed for the discrete dot-product unit. The carry-out from the bottom half addition is removed from the critical path. Furthermore, the floating-point multiplier has a compound addition with advanced normalization, round and sticky bit generation. The Dadda reduction tree for the floating-point multiplier has 7 reduction stages; the delay of one stage is approximately the same as the delay of a full adder. After the final stage, a 47 bit and a 46 bit number are obtained. The sum of the two numbers is the desired product. Figure 4.3 shows the Dadda dot diagram for the significand multiplier; the red rectangles show full-adder inputs and the purple rectangles indicate half-adder inputs.

For a sum-of-squares floating-point arithmetic unit, the multipliers are replaced with squarers; a folded squarer is selected [31]. Figure 4.4 illustrates the floating-point squarer unit. Its sign bit is always 0. The exponent unit does not need addition to produce the output exponent; a one bit left shifted copy of the input exponent is used. The folded squarer size is around half of the size of a multiplier tree. Figure 4.5 shows the 24 bit folded squarer dot diagram. The folded squarer matrix is simpler because the hidden bit is 1. AND products are not necessary for the white dots since the hidden bit is $1(X_{23})$. The squarer with Dadda reduction has two fewer full-adder steps than the conventional Dadda multiplier.

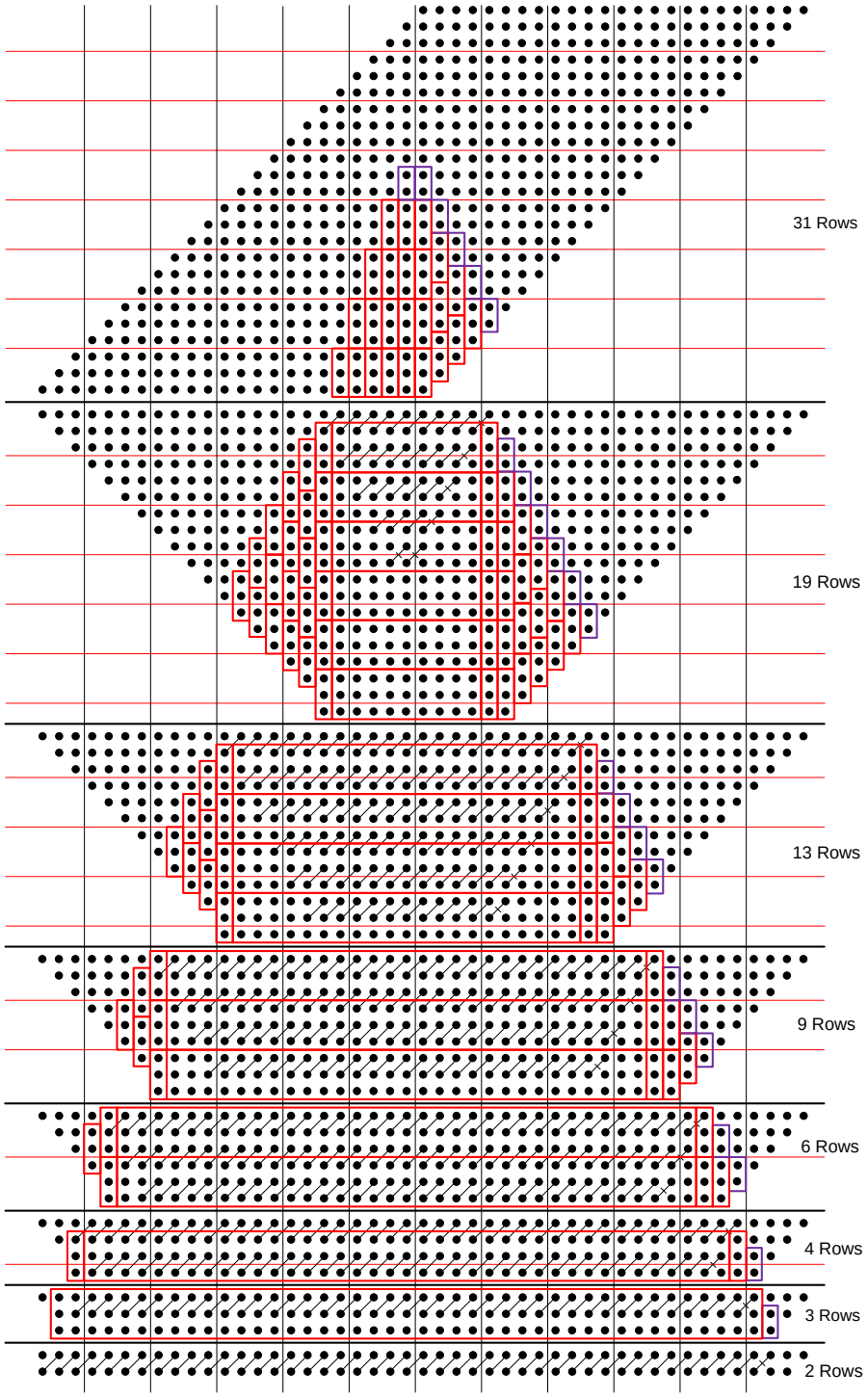


Figure 4.3: Dadda Dot Diagram for the Significand Multiplier

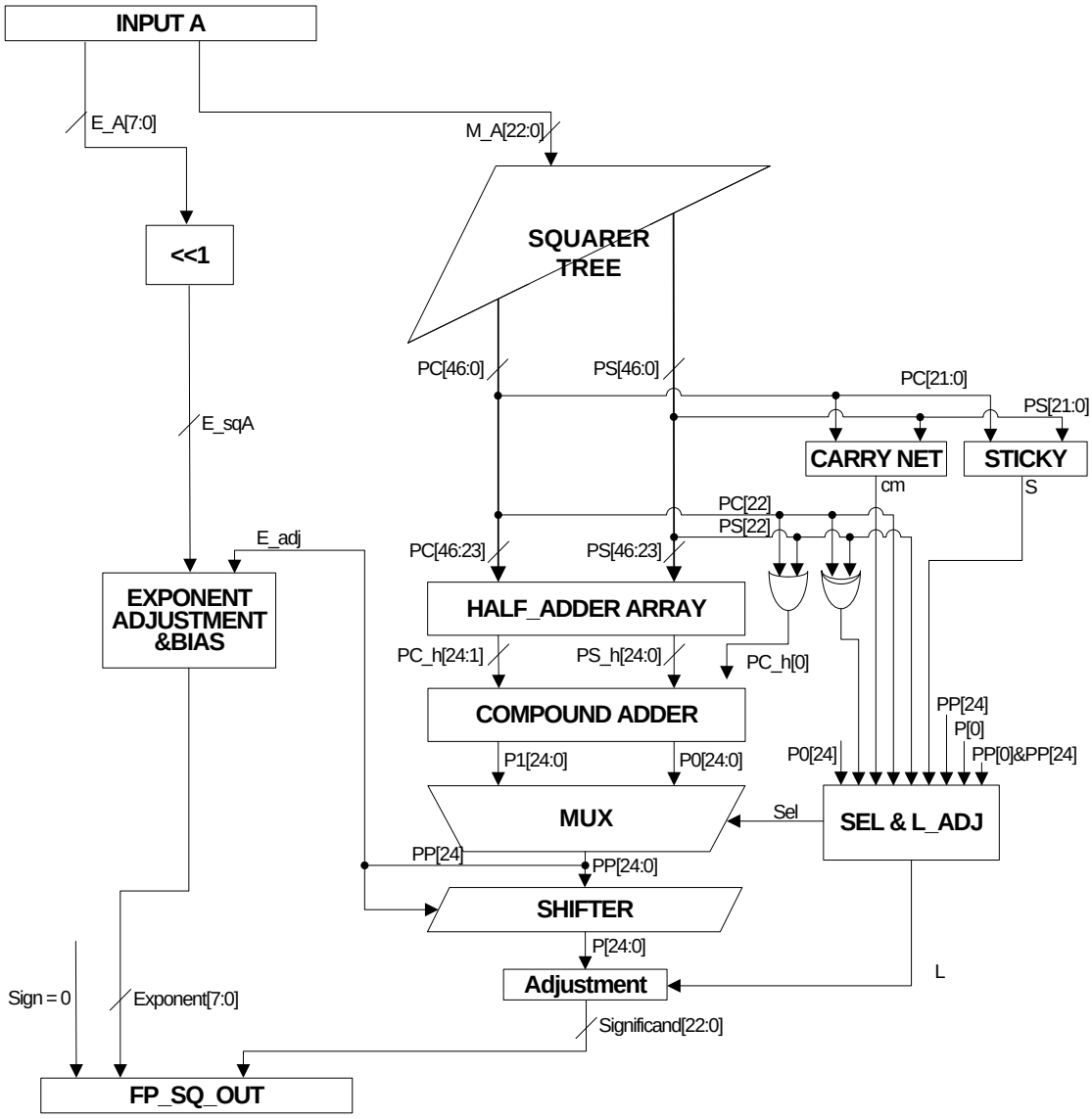


Figure 4.4: The Floating-Point Squarer

Significand squares = $(X_{23}\ X_{22}\ X_{21}\ X_{20}\ X_{19}\ X_{18}\ \ldots\ldots\ X_1\ X_0)^2$

$X_{23}X_{22}$	$X_{23}X_{21}$	$X_{23}X_{20}$	$X_{23}X_{19}$	$X_{23}X_{18}$	…..
X_{23}		$X_{22}X_{21}$	$X_{22}X_{20}$	$X_{22}X_{19}$	…..
		X_{22}	$X_{21}X_{20}$		…..
			X_{21}		…..

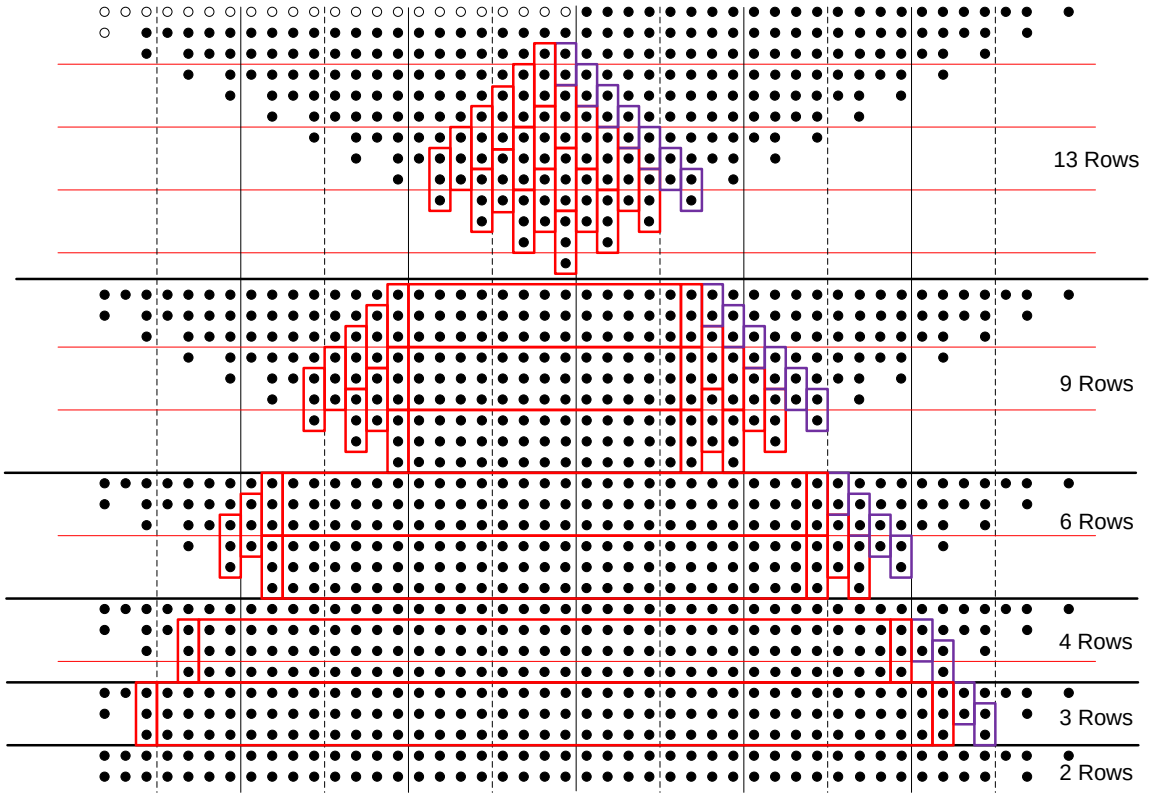


Figure 4.5: Dadda Dot Diagram for the Significand Squarer

FUSED TWO-TERM SUM-OF-SQUARES ARCHITECTURE

Various fused floating-point two-term sum-of-squares units are designed and introduced in this section. Designers can select one of these models based on the specifications and requirements of their system. The fused sum-of-squares units perform two square computations in parallel and add them without round/normalization; alignment can be performed before or after the significand square computations (pre/post-alignment model). By skipping the normalization and round steps, the fused sum-of-squares units have less power consumption, area, and delay than the discrete sum-of-squares units.

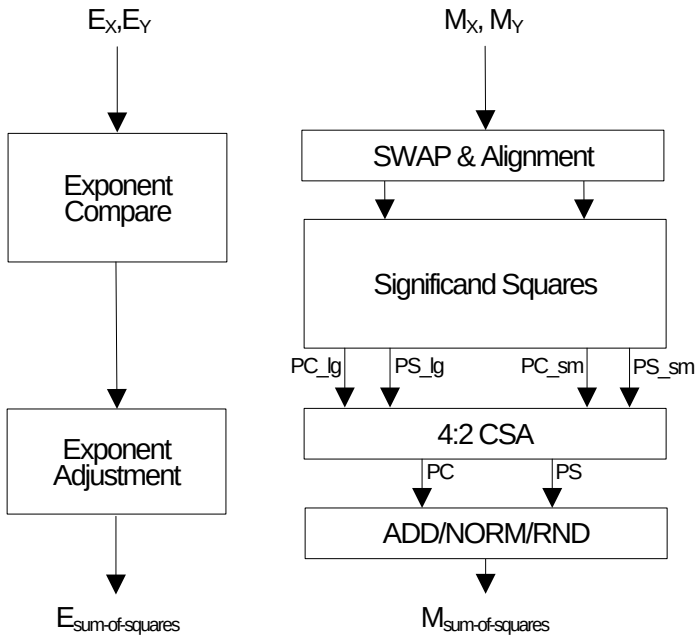


Figure 4.6: Basic Implementation of the Pre-Alignment Fused Sum-of-Squares Unit

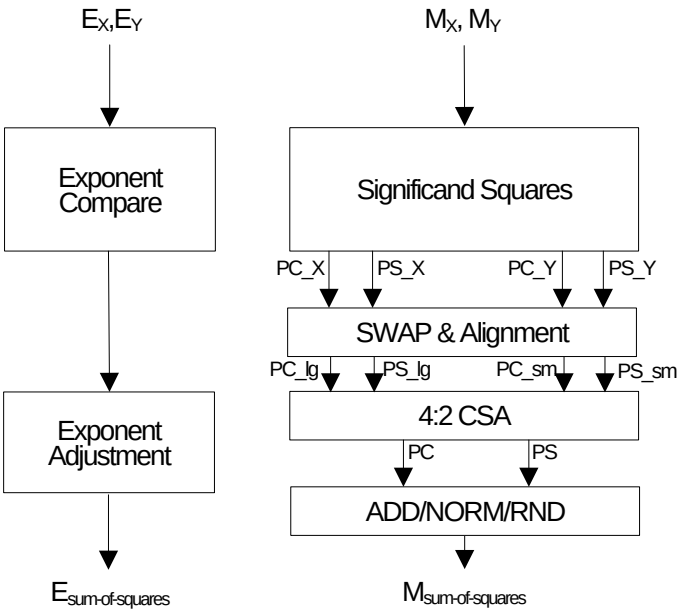


Figure 4.7: Basic Implementation of the Post-Alignment Fused Sum-of-Squares Unit

Figures 4.6 and 4.7 show the basic implementation of pre-alignment and post-alignment floating-point fused two-term sum-of-squares unit.

Width of Carry-Sum Processing

The two internal significand squarers each produce 47 bit carry and sum words, respectively. In Figures 4.6 and 4.7, PC_X, PC_Y, PC_lg, and PC_sm indicate the carry words while PS_X, PS_Y, PS_lg, and PS_sm indicate the sum words. The four carry and sum words are processed by the SWAP unit, two alignment units, and the 4:2 CSA for the post-alignment model. The four carry and sum words are processed by the 4:2 CSA for the pre-alignment model. In this dissertation, both of these processes are called the carry-sum processing. After the carry-sum processing, two words (PS and PC) are produced for the ADD/NORM/RND unit.

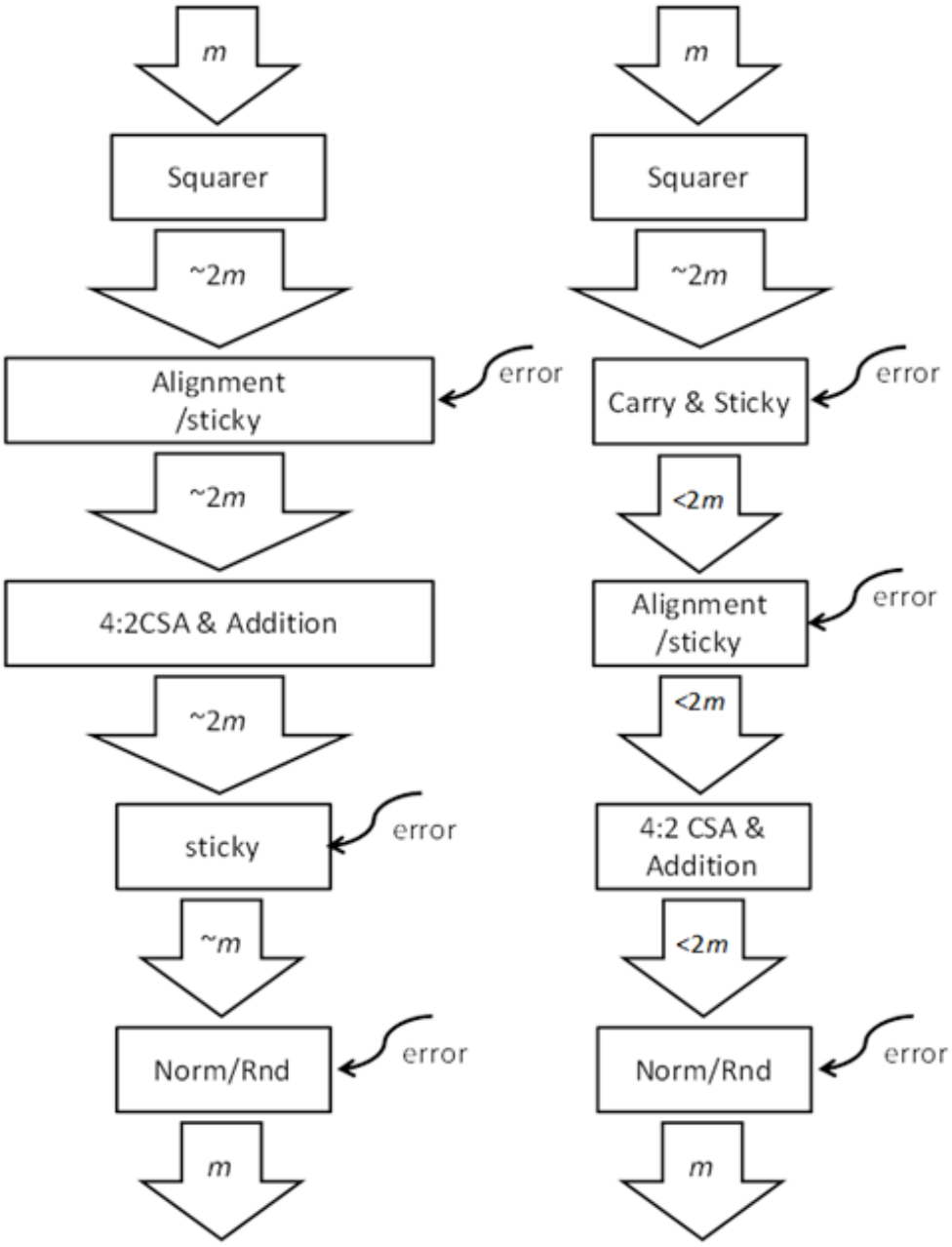


Figure 4.8: Full Width and Partial Width Post-Alignment Model

If the full width carry and sum words are used for the carry-sum processing, a 47 bit SWAP unit, two 47 bit alignment units, a 47 bit 4:2 CSA, and a 48 bit adder are required for the post-alignment model (a 47 bit 4:2 CSA, and a 48 bit adder are used for the pre-alignment model). If the reduced width carry and sum words are used for the carry-sum processing (e.g., partial width carry-sum processing), a partial bit-width SWAP unit, alignment unit, 4:2 CSA, and adder are employed for the post-alignment model (a partial width 4:2 CSA and adder are used for the pre-alignment model). Figure 4.8 shows the full width and partial width data-flow post-alignment models with the input width as *m*.

For the partial width model, the carry and sum networks that generate the carry bits and sticky bits from the least significant portion of the carry and sum words are required. For example, the partial width carry-sum processing model with 26-bit carry and sum word has carry bits and sticky bits generated from the bottom 21 bits of the carry and sum words. Figure 4.9 shows the carry-sum processing for the partial width post-alignment model. The structure of the carry and sticky net is same as the carry and sticky net of the floating-point multiplier as described in Chapter 1. The position of the carry bit is [lsb_top] (lsb+1 of CSA). Due to the position of the carry bit, the number of CSA inputs should be 6. However, the 6:2 CSA reduces performance and increases power consumption. To reduce the number of CSA inputs, the carry bit can be added in the lsb position twice.

The carry and sum word with smaller exponent are aligned based on the exponent difference. The two alignment units produce sticky bits (sticky_align_PS_Y and sticky_align_PC_Y).

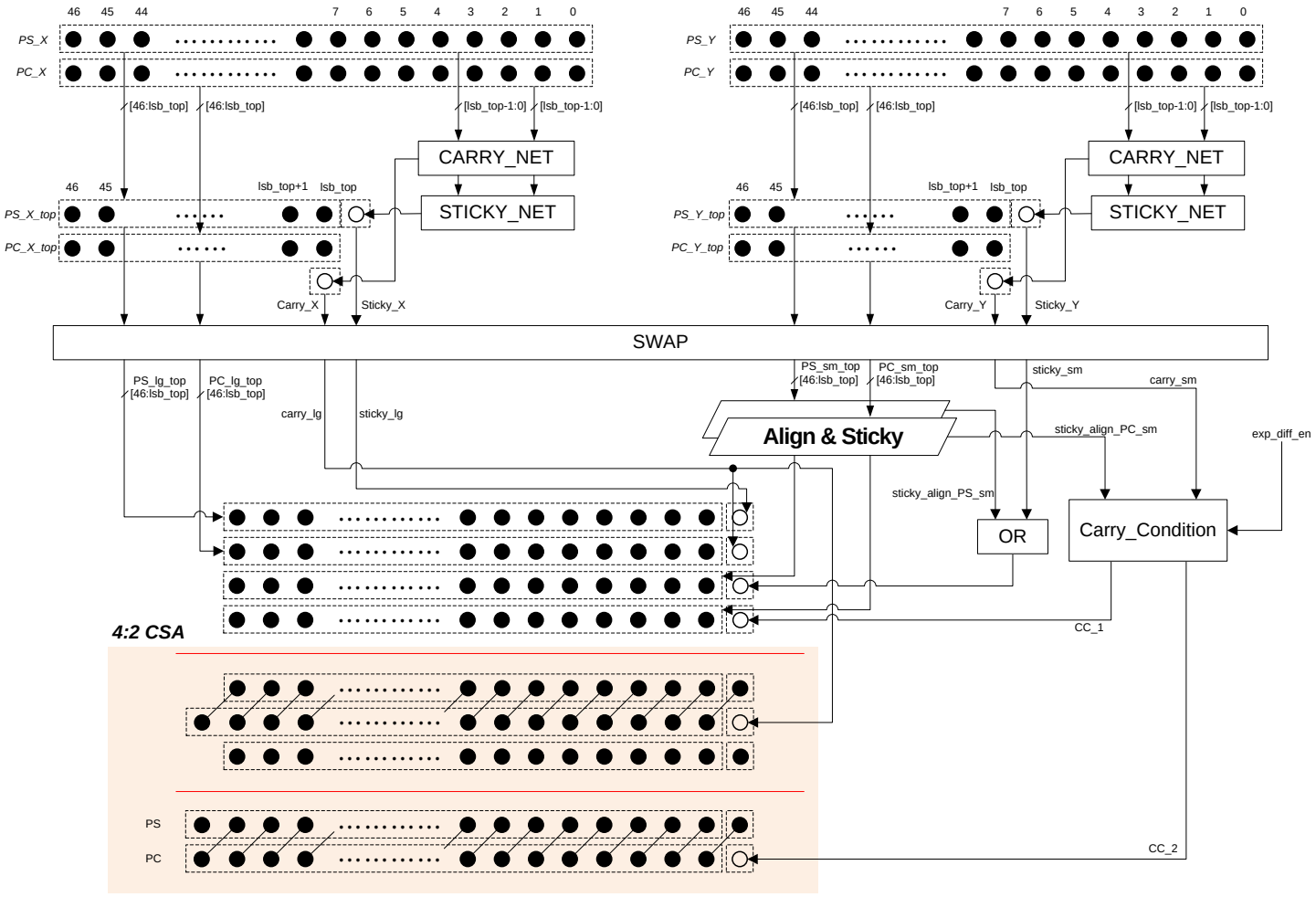


Figure 4.9: Carry-Sum for Partial Width Post-Alignment Model

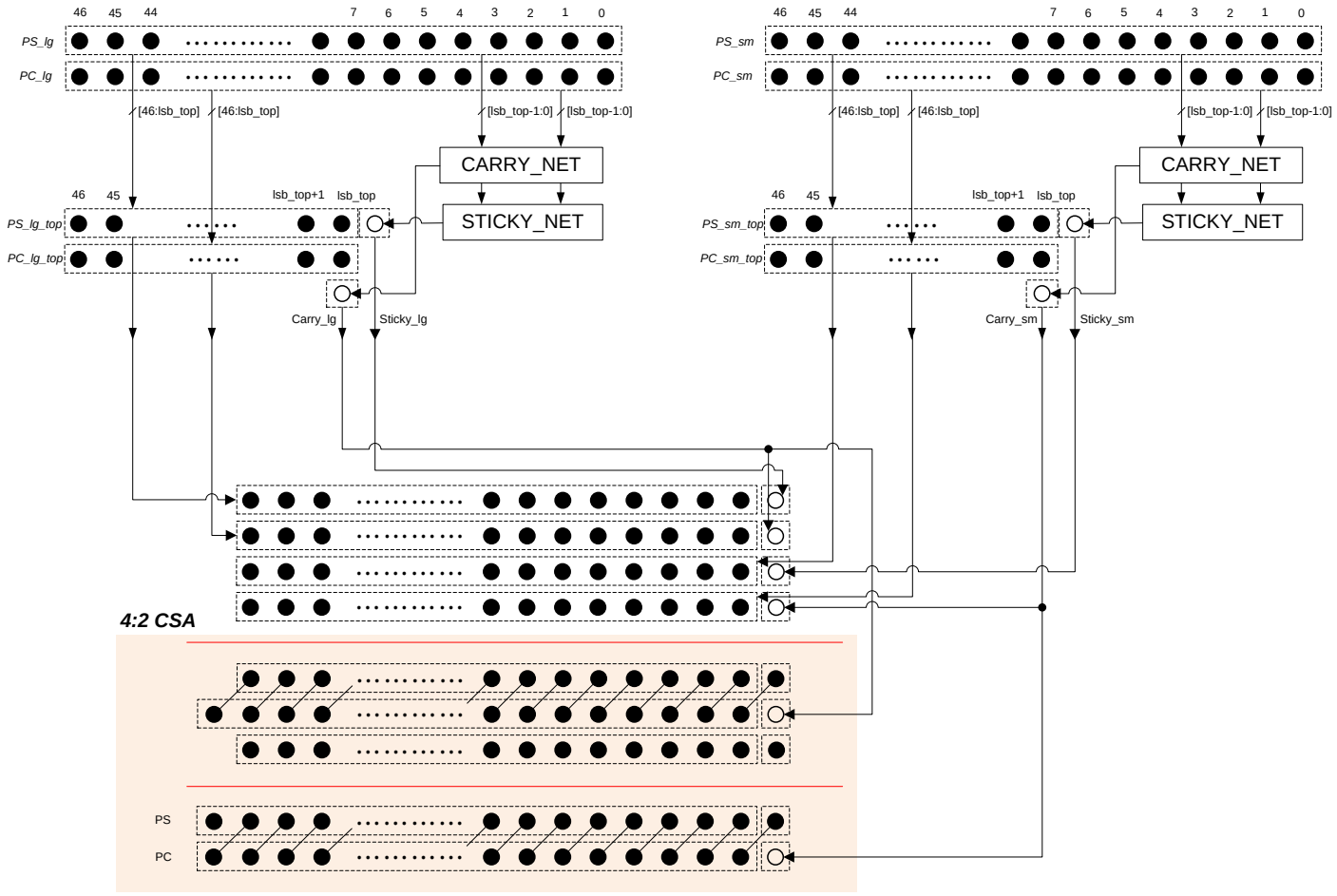


Figure 4.10: Carry-Sum for Partial Width Pre-Alignment Model

If the carry and sum word with the smaller exponent are aligned, the sticky_align_PS_sm bit will be *OR*ed with the sticky_sm bit from STICKY_NET and processed in the 4:2 CSA. In addition, the carry_sm bit from CARRY_NET is processed in Carry_Condition logic as shown Equations (4.1), (4.2), and (4.3).

exp_diff_en = (alignment) ? 1 : 0 (4.1)

CC1 = exp_diff_en ***AND*** sticky_align_PC_sm

OR NOT exp_diff_en ***AND*** carry_sm (4.2)

CC2 = ***NOT*** exp_diff_en ***AND*** carry_sm (4.3)

Figure 4.10 shows the carry-sum processing for the partial width pre-alignment model. The alignment is performed before the significand square computation so that the Carry-Condition logic is not required.

The width of the carry-sum processing affects the area, latency, and power-consumption. The partial width model has less power consumption, area, and latency. However, the precision is non-linearly proportional to the width of the carry-sum processing. Figures 4.11 and 4.12 show the absolute relative error of the fused sum-of-squares unit according to the width of the carry-sum processing. These figures also show the absolute relative error of the discrete sum-of-squares unit as a reference. Moreover, these figures indicate that the full width and large partial width models do not have enormous advantage in terms of precision; these two models only have large power consumption, area, and latency.

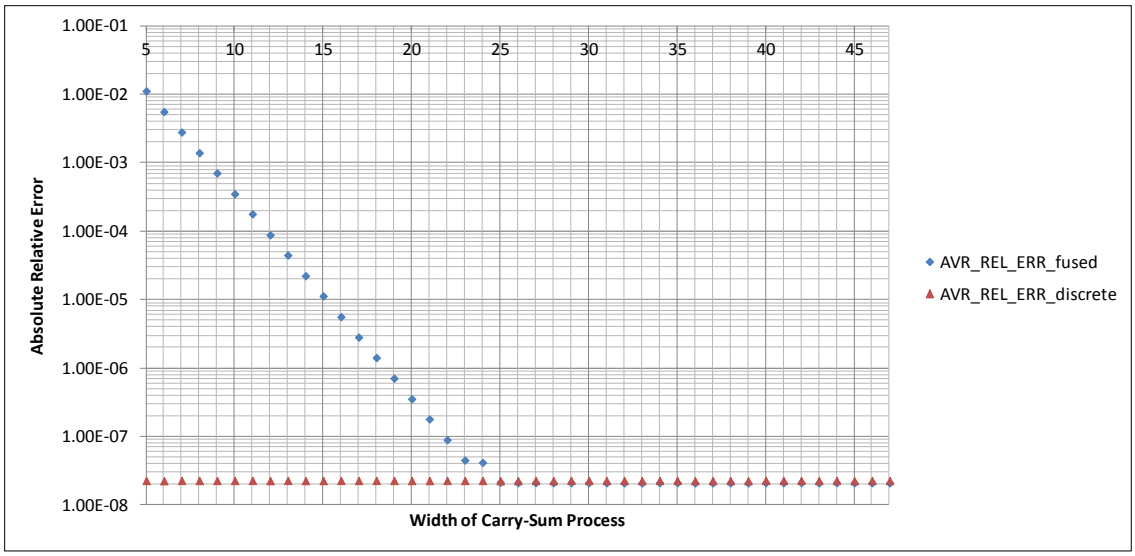


Figure 4.11: Absolute Relative Error According to Carry-Sum Processing Width

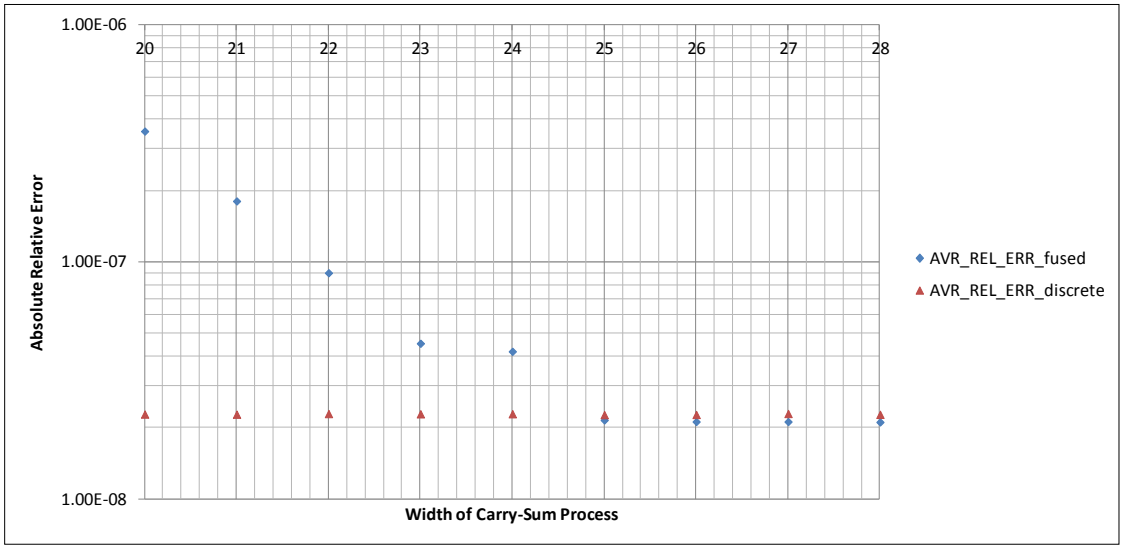


Figure 4.12: Magnified Graph from Figure 4.11

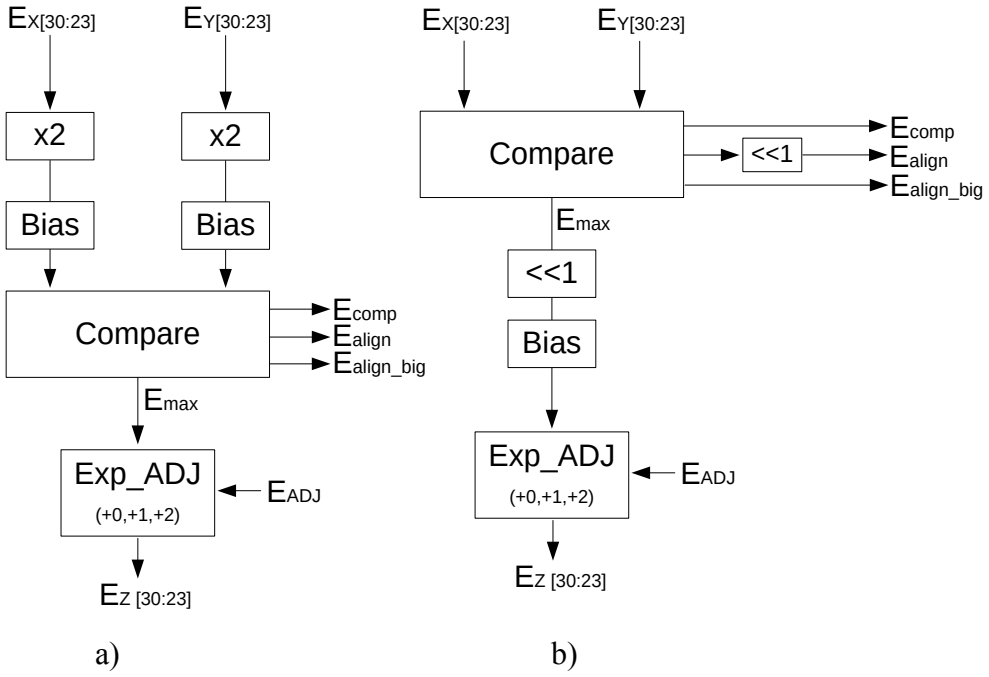


Figure 4.13: Exponent Unit for the Fused Sum-of-Squares Unit

Exponent Unit

The exponent unit can be designed as shown in Figure 4.13 (a). However, it can be simplified as shown in Figure 4.13 (b); it has one less bias unit which is a subtractor. The output bit-width of the compare unit for the alignment is 1 bit less.

Enhanced Compound Add/Round/Norm Unit

An enhanced fast-rounding system is proposed in this section. Table 4.1 shows the possible rounding cases for the fused sum-of-squares with post-alignment and partial carry-sum processing. The dots in Table 4.1 represent 1 or 0.

Table 4.1: Possible Rounding Cases for the Fused Sum-of-Squares Model with Post-Alignment and Partial Carry-Sum Processing

After SWAP	
PC_lg	001 . ●●●●●●●●…●●●
PS_lg	00● . ●●●●●●●●…●●●
PC_sm	001 . ●●●●●●●●…●●● (>> shift)
PS_sm	00● . ●●●●●●●●…●●● (>> shift)
After Significand Addition (PC + PS)	
PC + PS+ Round	1●● . ●●●●●●●●…●●● + 100 000
	01● . ●●●●●●●●…●●● + 10 000
	001 . ●●●●●●●●…●●● + 1 000

The possible integers of the result of the significand addition are from 1 to 7. The rounding can be performed before normalization. The shifted round bit is added based on the integer bit. With the possible rounding cases, an initial version of the fast-rounding system can be designed. Three 3-input adders (for rounded results) and one 2-input adder (for non-rounded results) are implemented in parallel for the initial version of the compound add/round/norm unit. The correct result is selected with a multiplexer and normalized.

A new advanced compound add/round/norm unit is proposed to enhance the performance and power consumption. The advanced fast round system needs only one 2-input adder and one 3-input adder. Table 4.2 shows all the possible scenarios for the new add/round/norm system. Scenarios #1 to #7 show the cases when rounding is requested. For scenarios #8, #9, and #10, the rounding addition is not necessary.

Table 4.2: Addition/Round/Normalization Scenarios

No_rnd_result = Carry + Sum (2-input addition) = 1.x x x x x x x x x U_2 U_1 U_0 G R S **Added4_result** = Carry + Sum + 1 0 0 0 0 0 (3-input addition) = 1.x x x x x x x x x U_2 U_1 U_0 G R S + 1 0 0 0 0 0
Scenario #1 Result of Significand addition = 1.xxxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U_1 U_0 = 0 0 *Valid result* = **No_rnd_result** with inverted $\mathbf{U_0}$ (U_1 U_0 : 0 0 → 0 1)
Scenario #2 Result of Significand addition = 1.xxxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U_1 U_0 = 0 1 *Valid result* = **No_rnd_result** with inverted $\mathbf{U_1}$,$\mathbf{U_0}$ (U_1 U_0 : 0 1 → 1 0)
Scenario #3 Result of Significand addition = 1.xxxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U_1 U_0 = 1 0 *Valid result* = **No_rnd_result** with inverted $\mathbf{U_0}$ (U_1 U_0 : 1 0 → 1 1)
Scenario #4 Result of Significand addition = 1.xxxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U1 U0 = 1 1 *Valid result* = **Added4_result** with inverted $\mathbf{U_1}$,$\mathbf{U_0}$ (U_1 U_0 : 1 1 → 0 0)
Scenario #5 Result of Significand addition = 1x.xxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U_1 = 0 *Valid result* = **No_rnd_result** with inverted $\mathbf{U_1}$ (U_1 : 0 → 1), then >> 1
Scenario #6 Result of Significand addition = 1x.xxxxxxxx U_2 U_1 U_0 G R S Rounding is requested & U_1 = 1 *Valid result* = **Added4_result** with inverted $\mathbf{U_1}$ (U_1 : 1 → 0), then >> 1
Scenario #7 Result of Significand addition = 1xx.xxxxxxx U_2 U_1 U_0 G R S Rounding is requested *Valid result* = **Added4_result** >>2
Scenario #8 , #9 and #10 No rounding is requested *Valid result* = **no_rnd_result** (*#8,* 1. xx…xxxGRS) Shifted right **no_rnd_result** by one (*#9,* 1x. xx…xxxGRS) Shifted right **no_rnd_result** by two (*#10,* 1xx. xx…xxxGRS)

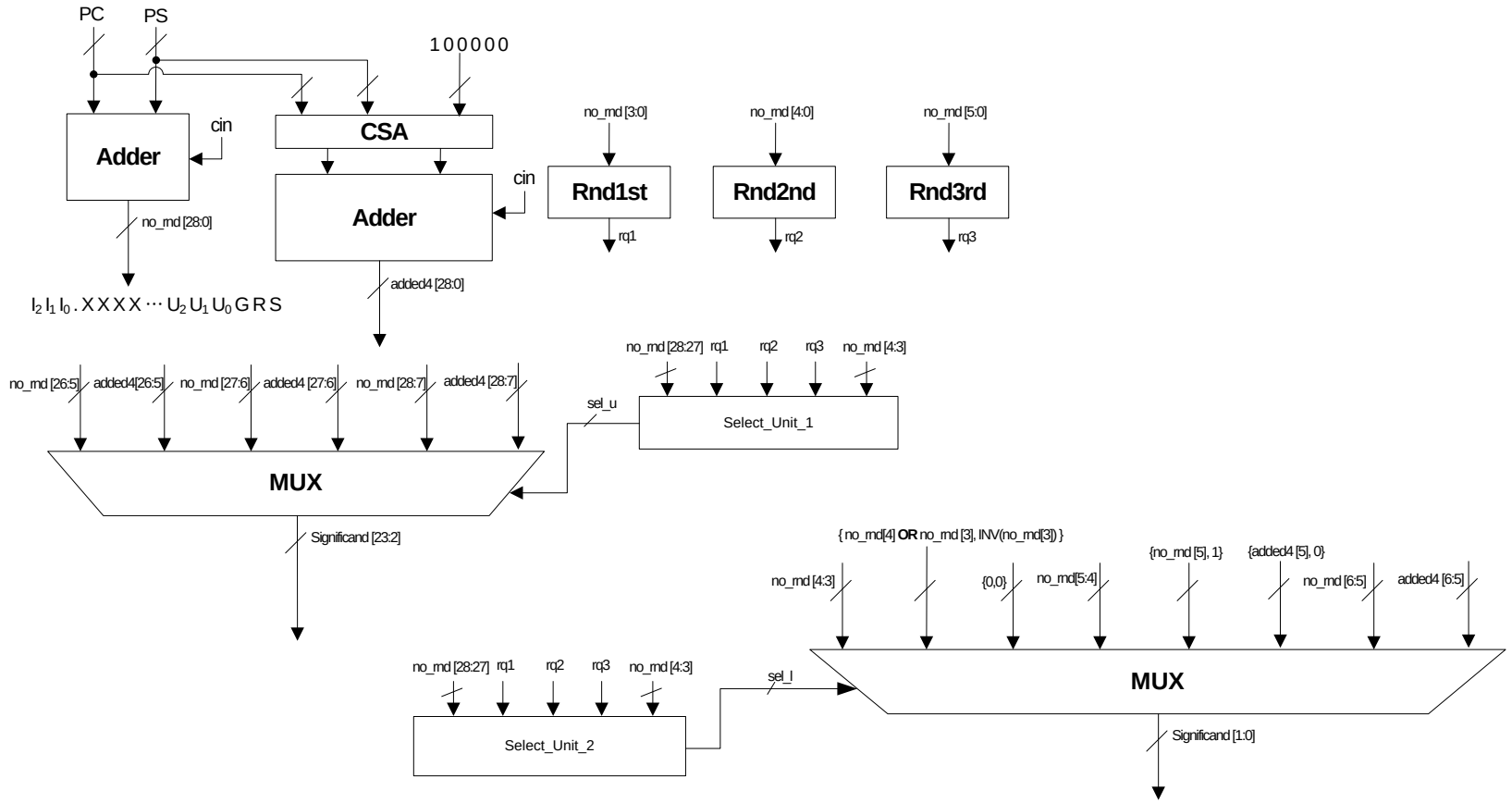


Figure 4.14: Block Diagram of the Compound Add/Round/Norm Unit

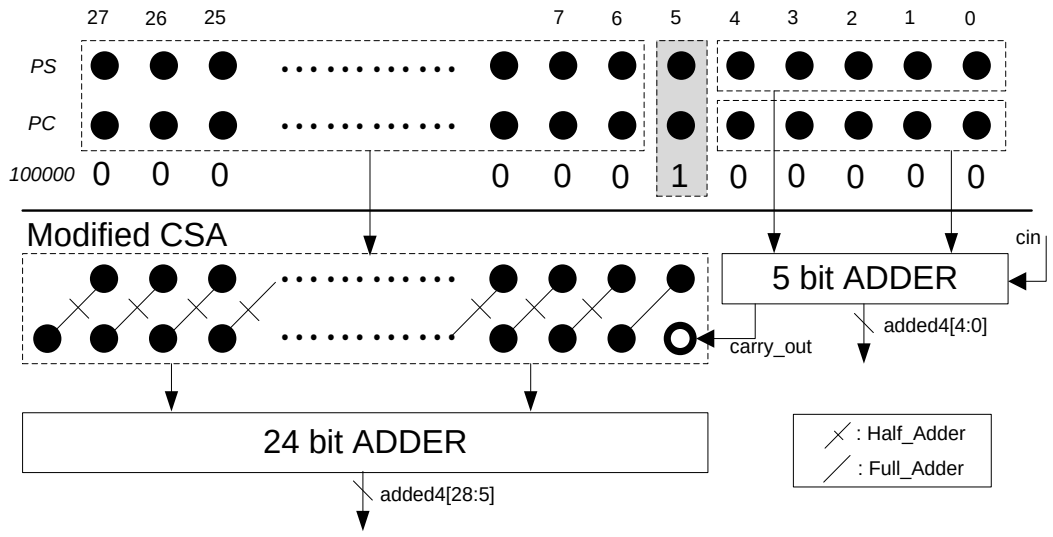


Figure 4.15: Block Diagram of the Significand Adder for the added4 Signal

Figure 4.14 shows a block diagram of the compound add/round/norm unit with 26 bit width carry-sum processing. The three round request units (Rnd1st, Rnd2nd, and Rnd3rd) are performed in parallel for the three cases (Scenario #1-#4, Scenario #5-#6, and Scenario #7).

For the 3-input adder, a CSA is used instead of a series of two normal adders to reduce the latency. The 3-input significand addition for the added4 signal can be designed with a modified CSA, a 5-bit adder, and a 24-bit adder for better performance and less area and power consumption. Figure 4.15 shows the block diagram of the significand adder for the added4 signal. The modified CSA consists of one full adder and twenty two half adders because the third addend is a constant (10000_2). The constant has only one 1; the one full adder is used for this 1. For the addition of the bottom 5 bits of the sum and

carry signal, a 5-bit adder is used. The carry-out from the 5-bit adder is added in the following 24 bit adder. The carry-out of the 24 bit adder is not used because the maximum result of the significand addition with rounding needs 29-bit. These split adders have less delay because the modified CSA and 5-bit adder perform in parallel.

Select_Unit_1 selects the valid top 22 bits of the significand and Select_Unit_2 selects the valid bottom 2 bits of the significand based on integer cases (I_2 and I_1), round request signals (rq1, rq2, and rq3), U_1, and U_0. Table 4.3 shows the selection table for Select_Unit_1 and Select_Unit_2. Scenarios #1, #2, #3, and #8 shown in Table 4.2 can be merged and Scenarios #5 and #9 can be combined for the top 22 bits of the significand. The first row of the Select_Unit_1 section represents Scenario #8 and the second row of the Select_Unit_1 section corresponds to #1, #2, and #3. The fourth and fifth rows of the Select_Unit_1 section represent Scenarios #5 and #9, respectively. A 22 bit 6-input multiplexer is used for the top 22 bits based on the merged scenarios. For the bottom 2 bits, Scenarios #1, #2, and #3 can be merged because the U_1 and U_0 of the valid result can be implemented by *OR*(U_1,U_0) and *INV*(U_0), respectively. The second row of the Select_Unit_2 section represents #1, #2, and #3.

To check for overflow caused by rounding, the leading ones of no_rnd[28:27] and added4[28:27] are compared when the added4 signal is valid. If they are different, overflow has occurred.

Table 4.3: Selection Tables for Compound add/round/norm

Select_Unit_1							
I_2	**I_1**	**rq1**	**rq2**	**rq3**	**U_0•U_1**	**U_1**	**Significand[23:2]**
0	0	0	x	x	x	x	no_rnd[26:5] (merged *)
0	0	1	x	x	0	x	no_rnd[26:5] (merged *)
0	0	1	x	x	1	x	added4[26:5]
0	1	x	0	x	x	x	no_rnd[27:6] (merged **)
0	1	x	1	x	0	0	no_rnd[27:6] (merged **)
0	1	x	1	x	x	1	added4[27:6]
1	x	x	x	0	x	x	no_rnd[28:7]
1	x	x	x	1	x	x	added4[28:7]

Select_Unit_2								
I_2	**I_1**	**rq1**	**rq2**	**rq3**	**U_0•U_1**	**U_1**	**Significand[1]**	**Significand[0]**
0	0	0	x	x	x	x	no_rnd[4]	no_rnd[3]
0	0	1	x	x	0	x	*OR*(no_rnd[3],[4])	*INV*(no_rnd[3]**)**
0	0	1	x	x	1	x	0	0
0	1	x	0	x	x	x	no_rnd[5]	no_rnd[4]
0	1	x	1	x	0	0	no_rnd[5]	1
0	1	x	1	x	x	1	added4[5]	0
1	x	x	x	0	x	x	no_rnd[6]	no_rnd[5]
1	x	x	x	1	x	x	added4[6]	added4[5]

IMPLEMENTATION OF THE FUSED SUM-OF-SQUARES UNIT

For the width of the carry-sum processing, a 26 bit width is proposed because it has better precision than that of the discrete model. Full width model and larger models are not proposed because these models have larger power consumption, area, and latency without a significant advantage in terms of precision compared to the 26 bit width model. A 25 bit width model also can be a good option to designers because the precision of the 25 bit width model is still better than that of the discrete model.

Fused Floating-Point Sum-of-Squares Unit with Post-Alignment

The carry and sum words from the squared significands are swapped if necessary based on the comparison of the two exponents. The carry and sum words with the smaller exponent are aligned by two one-direction-alignment units. Figure 4.16 shows the block diagram of the fused sum-of-squares unit with the post-alignment unit and 26 bit width carry-sum processing. The advanced fast rounding system can be implemented for the addition, normalization, round, and post-normalization unit. The sum-of-squares computation only performs addition. Therefore, the sub-modules related to subtraction are not required. Also a sign-bit decision unit is also unnecessary.

Fused Floating-Point Sum-of-Squares Unit with Pre-Alignment

Alignment can be performed before the significand square computation. The pre-alignment model requires a single alignment unit while the post-alignment model requires two alignment units. The number of multiplexers in the SWAP unit in the pre-

alignment model is less compared with the post-alignment model. Furthermore, the SWAP unit of the pre-alignment model is smaller than that of the post-alignment unit because the bit-width of the significand is increased after the significand square computation. However, the alignment unit needs the valid exponent difference from the exponent unit, which increases the latency. In addition, the power consumption of the significand squarers is increased because the hidden bit is moving by the alignment and the AND array of the significand squarers for the hidden bit cannot be eliminated in the pre-alignment model. As a result, the power consumption is not significantly improved. Figure 4.17 shows the block-diagram of the fused floating-point sum-of-squares unit with the pre-alignment unit.

Usage for General Purpose Floating-Point Squarer

The fused floating-point two-term sum-of-squares unit can be used for general purpose floating-point square computation. The shortest path for square computation passes by the significand squarer for the input X, Carry_Net, 4:2 CSA, and the compound add/rnd/norm unit. Therefore, the input X is used for the square computation (X^2) to reduce the latency. To enhance the performance, a bypass unit can be used. Figure 4.18 shows the fused floating-point two-term sum-of-squares unit with the bypass unit. The signal OP_GEN enables the floating-point square computation. In addition, the input Y is set to zero. With the bypass process, the critical path is on the significand squarer for the input X, Carry_Net, MUX, and the compound add/rnd/norm unit for the square computation. However, the bypass unit increases the latency for the sum-of-squares computation.

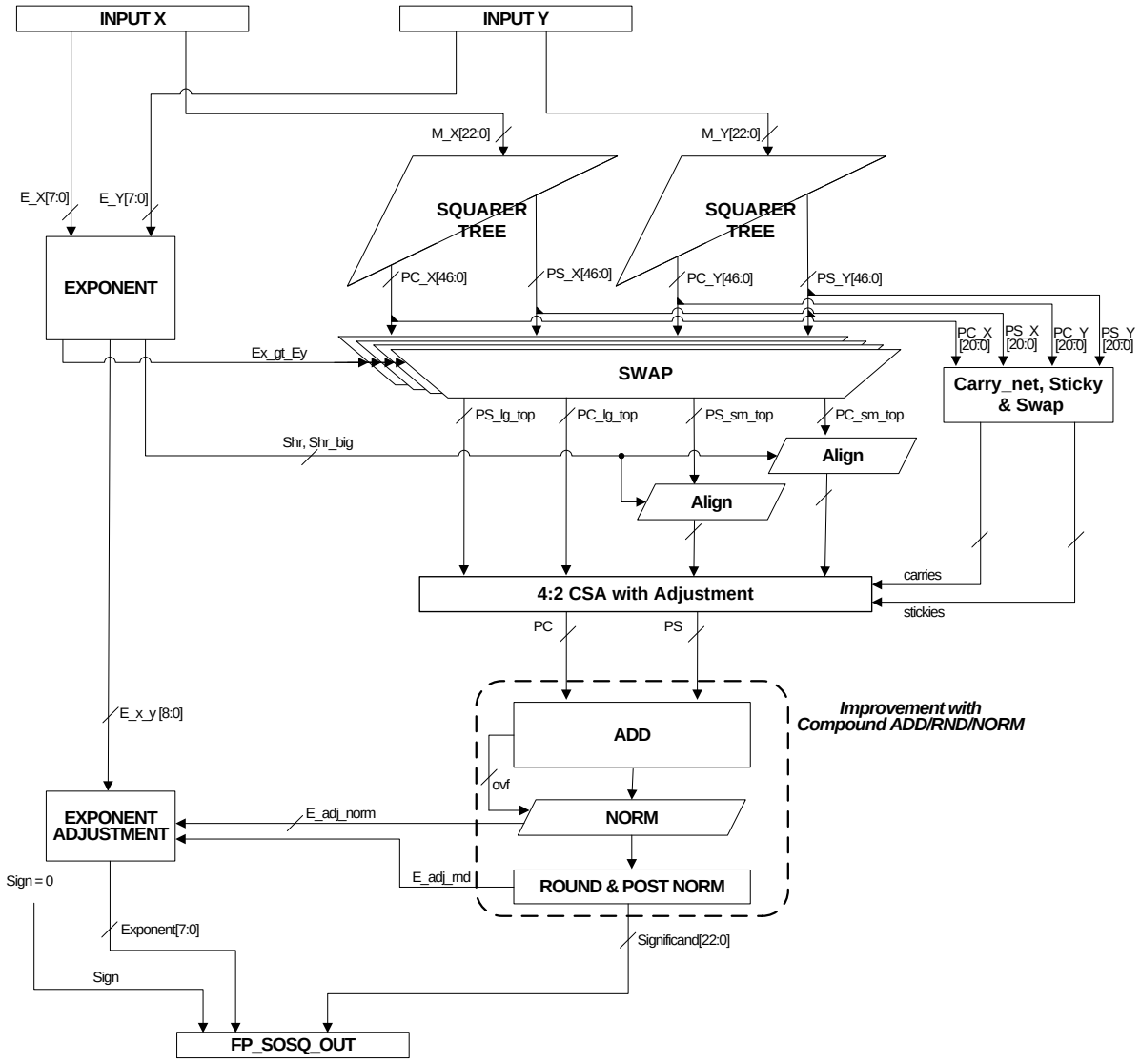


Figure 4.16: Fused Floating-Point Sum-of-Squares Unit with Partial Width Post-Alignment Model

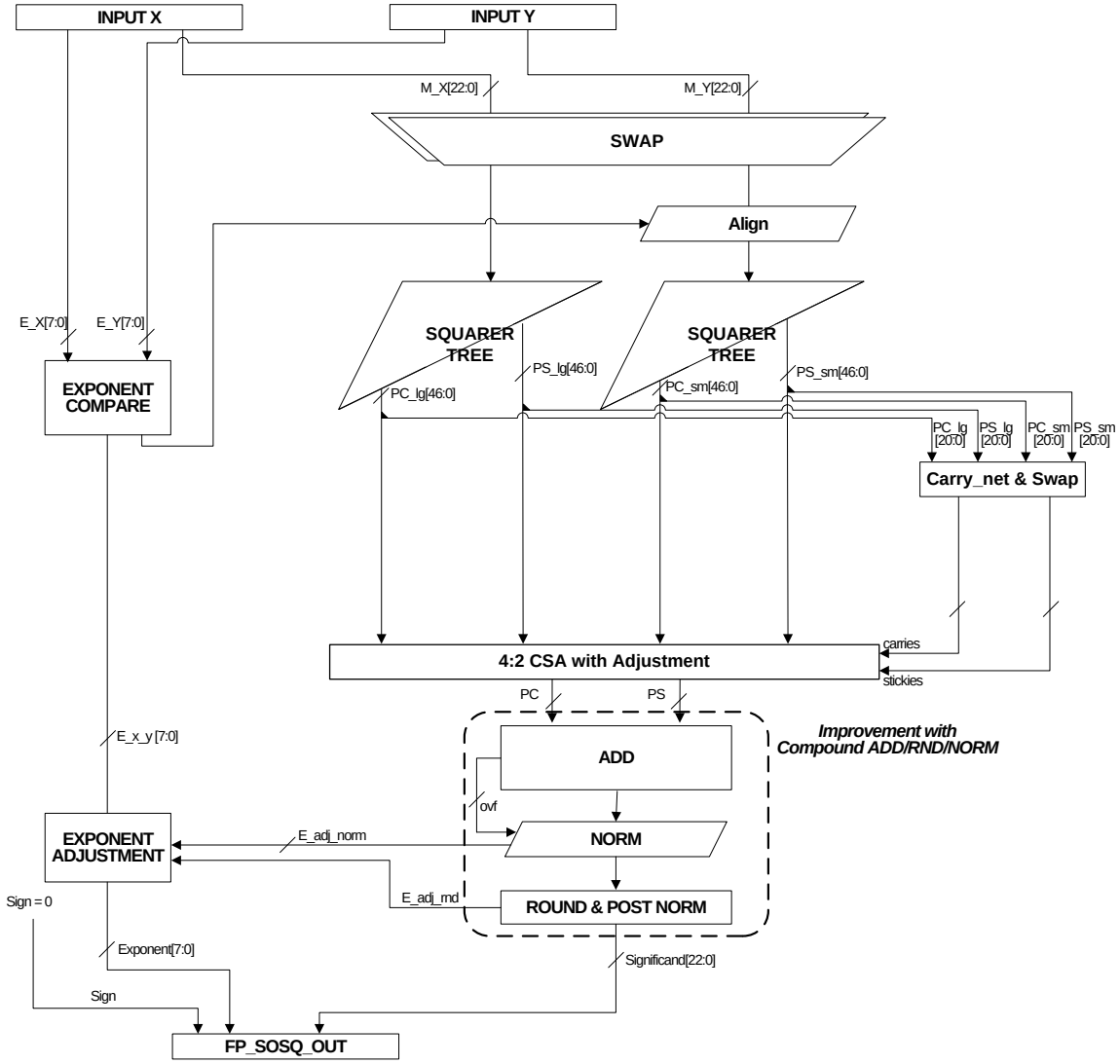


Figure 4.17: Fused Floating-Point Sum-of-Squares Unit with Partial Width Pre-Alignment Model

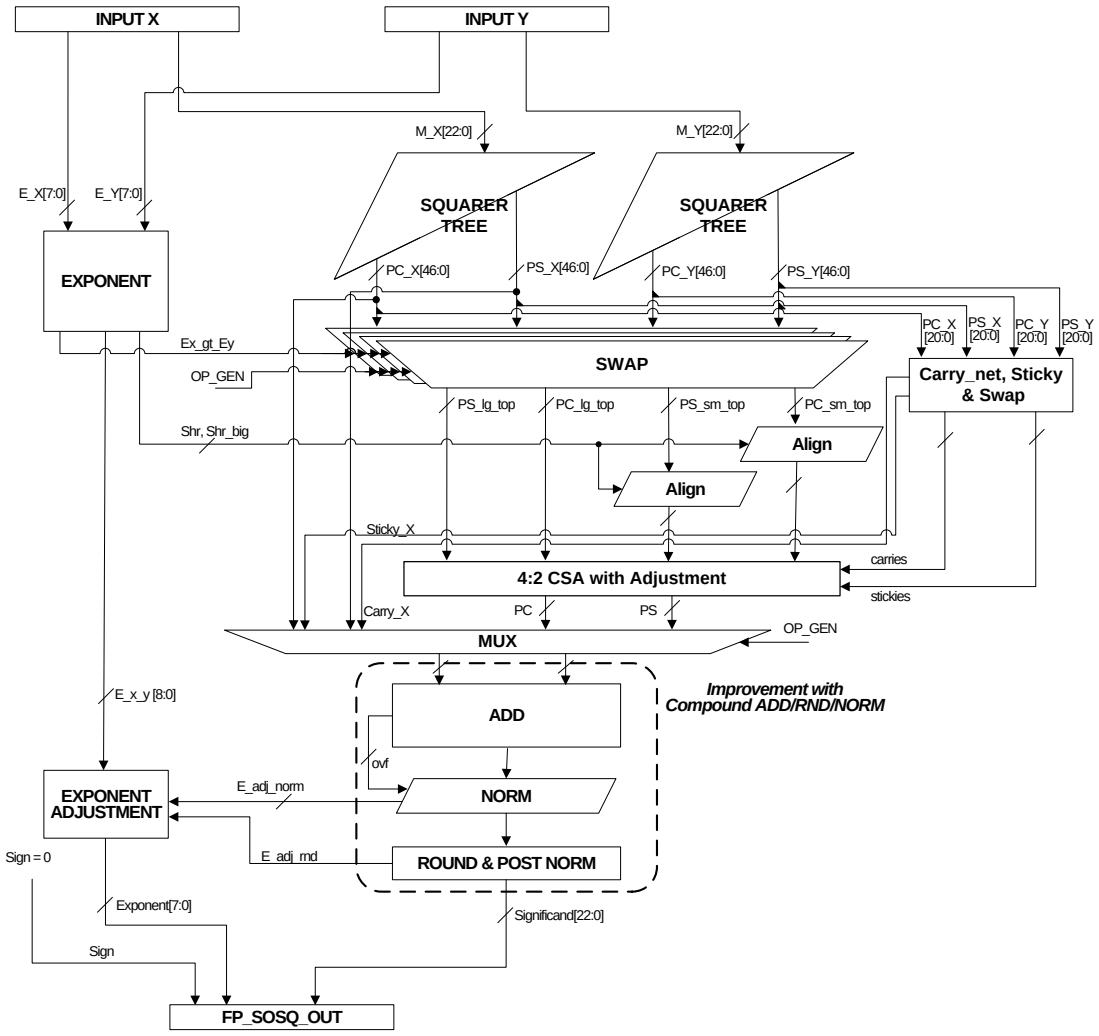


Figure 4.18: Fused Sum-of-Squares Unit with the Bypass for Floating-Point Square Computation

SIMULATION RESULTS

To fully understand the fused sum-of-squares unit, the floating-point dot-product units and the floating-point sum-of-squares units are implemented (non-pipelined models). The designs were optimized and synthesized for the best performance and compared under the same voltage, temperature, and frequency conditions. To show how much the fused architecture affects the area, latency, and power consumption, the discrete sum-of-squares unit with a half floating-point adder that only performs addition is implemented; the half floating-point adder does not have sub-modules related to subtraction such as Leading Zero Anticipator (LZA), 2's complement, and sign-decision unit. The discrete units and fused units were implemented and simulated by Synopsys and Cadence CAD tools as described in Chapter 2.

Figure 4.19 shows the lay-out (95um x 95um) of the fused floating-point sum-of-squares unit with the compound add/round/norm unit. The core density without physical cells is 74.5%. Table 4.4 shows the cell area, power consumption (at 150MHz), and latency for the dot product and sum-of-squares units. Figure 4.20 shows the relative ratios of the area, power consumption, and latency relative to the fused sum-of-squares unit with compound add/rnd/norm, partial carry-sum processing, and post-alignment (COMP, PCSP, POA).

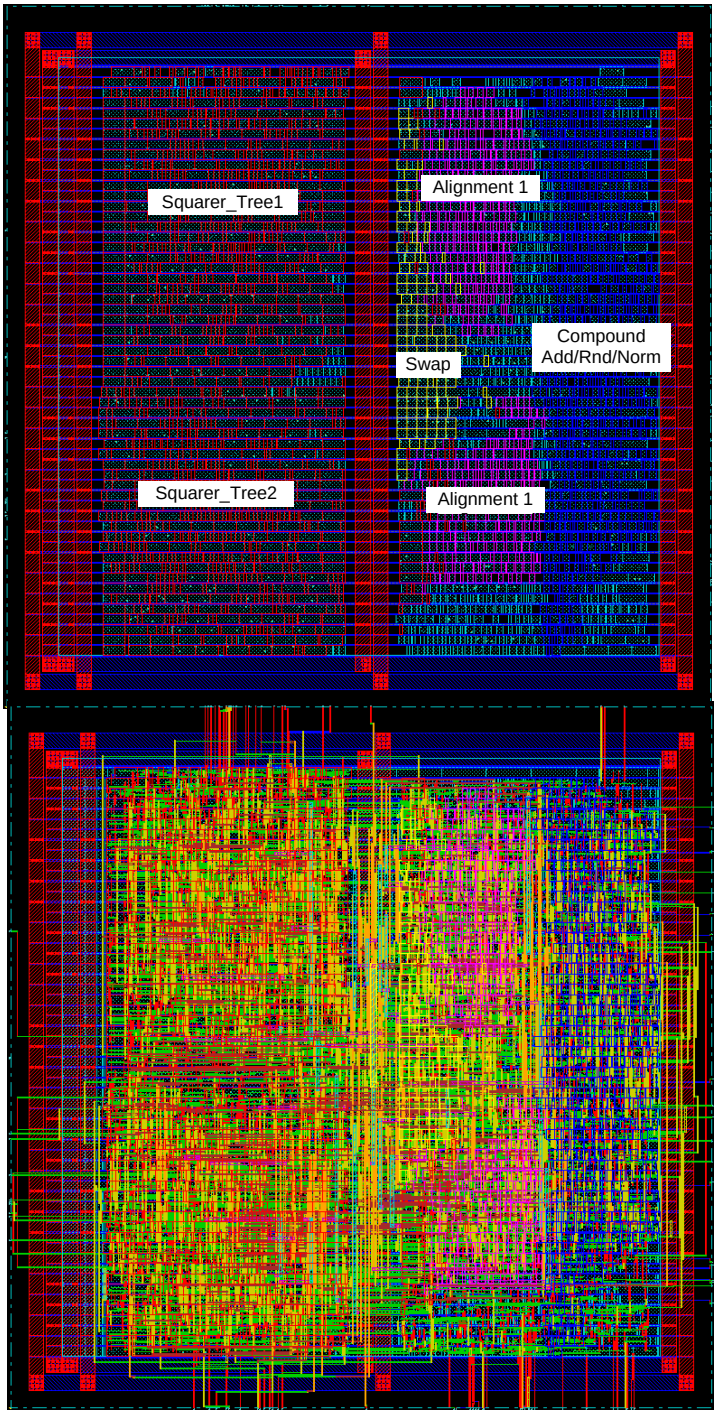


Figure 4.19: Layout of the Fused Sum-of-Squares Unit with Compound ADD/RND/NORM, Partial Carry-Sum Processing, and Post-Alignment

Table 4.4: Comparison of the Floating-Point DP and SoSQ Units

Unit Type	Standard Cell Area (μm^2)	Total Power (mW)	Optimized Latency (ns)
Discrete SRL DP	6,050 (126%)	1.63 (168%)	8.61 (297%)
Discrete PAR DP	9,501 (197%)	2.50 (258%)	6.10 (201%)
Discrete SRL SoSQ	4,419 (92%)	1.09 (112%)	7.74 (267%)
Discrete SRL SoSQ(half)	3,205 (67%)	0.77 (79%)	6.15 (212%)
Discrete PAR SoSQ	6,237 (129%)	1.57 (162%)	5.64 (194%)
Discrete PAR SoSQ(half)	5,023 (104%)	1.25 (129%)	4.03 (139%)
Fused SoSQ (PCSP, POA)	4,667 (97%)	0.93 (96%)	3.18 (110%)
Fused SoSQ (COMP, FCSP, POA)	5,647 (117%)	1.11 (114%)	3.02 (104%)
Fused SoSQ (COMP, PCSP, PRA)	4,572 (95%)	1.00 (103%)	3.45 (119%)
Fused SoSQ (COMP, PCSP, POA)	4,817 (100%)	0.97 (100%)	2.90 (100%)

*Compound ADD/RND/NORM: COMP

*Partial Carry-Sum Processing: PCSP

*Full Carry-Sum Processing: FCSP

*Post-Alignment: POA

*Pre-Alignment: PRA

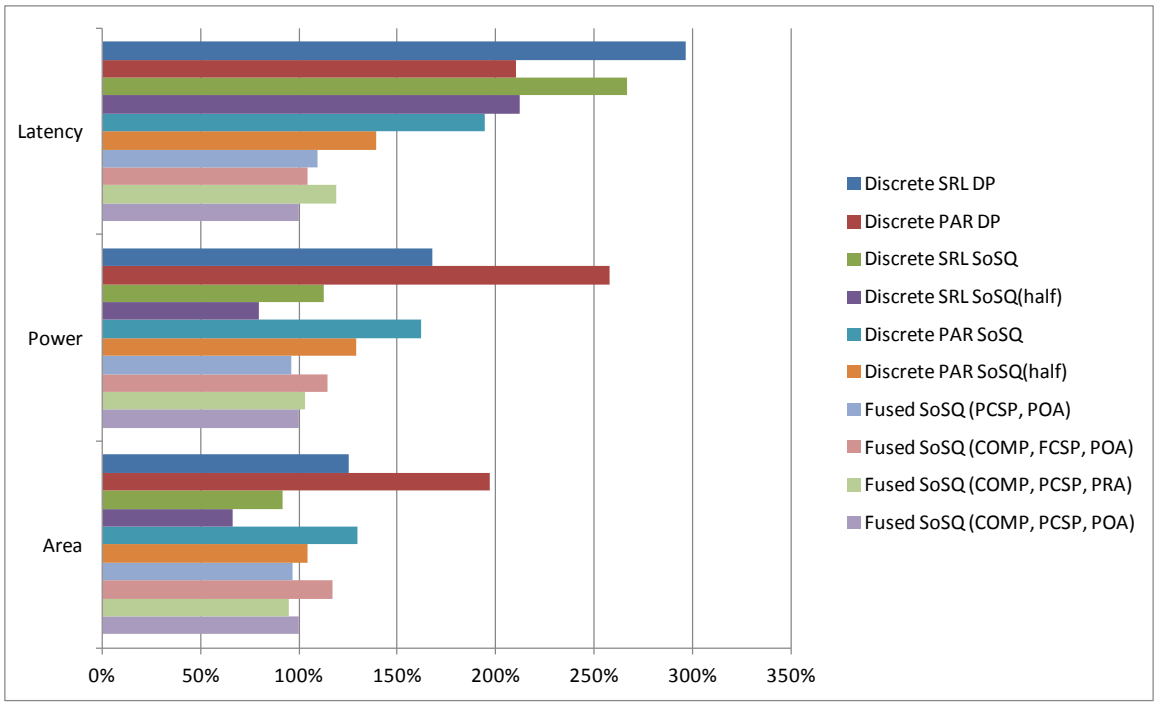


Figure 4.20: Relative Ratio of Performance Figures

Table 4.5: Critical Path of the Fused Sum-of-Squares unit with Compound ADD/RND/NORM, Partial Carry-Sum Processing, and Post-Alignment

Unit Type	Latency (ns)	Percentage
Significand Squarer	0.81	28%
Swap	0.07	2%
Alignment	0.38	13%
CSA	0.27	9%
Compound Addition	0.87	30%
Exponent Adjustment	0.24	8%
Exception	0.26	9%
Total	2.90	100%

Compared to the discrete parallel sum-of-squares unit with the half functional addition, the area, power consumption, and latency of the fused sum-of-squares unit with COMP, PCSP, and POA are 4%, 22%, 28%. The area, power consumption, and latency of the fused sum-of-squares with COMP, PCSP, and POA are 23%, 38%, and 49% less than those of the discrete parallel sum-of-squares unit with a standard full adder.

The fused sum-of-squares unit with the advanced rounding system (COMP, PCSP, and POA) is 9% faster than the fused sum-of-squares unit with the normal rounding system (PCSP and POA), but has 3% and 5% greater power consumption and area. Compared to the fused sum-of-squares unit with the full width carry-sum processing (COMP, FCSP, and POA), the fused sum-of-squares with COMP, PCSP, and POA has 15%, 13%, and 4% less area, power consumption, and latency. The full width carry-sum processing model does not have an advantage in terms of precision as described in the previous section. The fused sum-of-squares unit with pre-alignment has only 5% less area compared to the fused sum-of-squares unit with post-alignment. However, the latency is increased by 16% compared to the proposed fused unit.

The sum-of-squares units have lower power consumption, area, and latency than the corresponding dot product units. It is because the size of a significand squarer is around half the size of a significand multiplier. In addition, the significand squarer has two fewer full adder steps than the significand multiplier in terms of latency. Furthermore, the area, power consumption, and latency of the significand addition of the fused sum-of-squares unit are less than those of the significand addition of the dot-product unit with a full functional adder because it does not implement the subtract function. Without the subtract function, the rounding and normalization systems also

become simpler and faster. Furthermore, the compound add/rnd/norm of the fused sum-of-squares unit reduces the latency. Table 4.5 shows the portion of the sub-module latency in the critical path with the extracted RC parameters for the fused sum-of-squares unit with post-alignment, 26 bit partial carry-sum process, and compound ADD/RND/NORM.

Table 4.6 shows the latency results of the fused sum-of-squares unit with/without the bypass unit for the square computation. The latency of the fused sum-of-squares unit with the bypass unit is 4% less than that of the normal fused floating-point unit for the square computation. However, for the sum-of-squares computation, the fused unit with the bypass unit has 6% larger latency than the normal fused unit.

Table 4.6: Area, Power Consumption, and Latency of the Floating-Point Multiplier, Squarer and Fused Sum-of-Squares Unit with Two Computations

Unit Type	Computation Type	Optimized Latency (ns)
Fused SoSQ with the bypass (COMP, PCSP, POA)	X^2	2.74
	$X^2 + Y^2$	3.07
Fused SoSQ (COMP, PCSP, POA)	X^2	2.86
	$X^2 + Y^2$	2.90
Floating-Point Squarer	X^2	2.03
Floating-Point Multiplier	$X \times X$	2.46

CHAPTER 5: FLOATING-POINT FUSED MAGNITUDE UNIT

OVERVIEW AND MOTIVATION

A magnitude operation can be used for many graphics and signal processing applications such as finding the magnitude of complex numbers, the magnitude of vectors, and conversion from rectangular to polar coordinates [32]. Radar signal processing needs computation of the magnitude of a complex sample stream [33]. In addition, many computer languages such as MATLAB, Mathematica, PASCAL, PHP, and Java (since ver.1.5), and C++ (with library) have instructions for the magnitude computation.

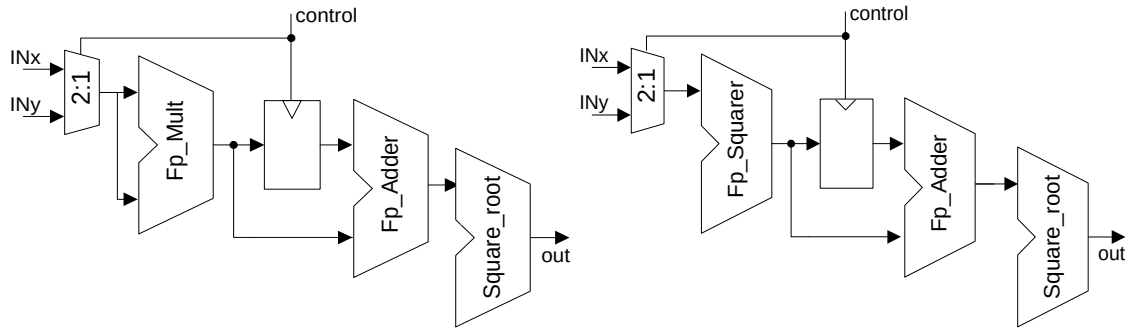
The goal of this chapter is to propose a new type of fused architecture for the magnitude computation ($\sqrt{X^2 + Y^2}$) with less power consumption, area, and latency than conventional floating-point units. Conventionally, the magnitude computation is executed by discrete serial or parallel floating-point units with conventional floating-point multiplication, squaring, addition, and square-root units. The proposed fused floating-point magnitude design has an enhanced exponent unit because the exponent processes of the sum-of-squares and square-root can be merged. Moreover, normalization to make the exponent an even number is not necessary for the fused magnitude unit. Normalization and rounding between the squares, addition and square-root computations are eliminated. A pipelined model for the fused magnitude unit is suggested.

This chapter compares the proposed fused magnitude unit with conventional discrete floating-point magnitude units. Compared with the discrete parallel magnitude

unit realized with conventional floating-point squarers, an adder, and a square-root unit, the fused floating-point magnitude unit has 24% less area, 24% less latency, and 27% less power consumption (26% less in the pipeline model).

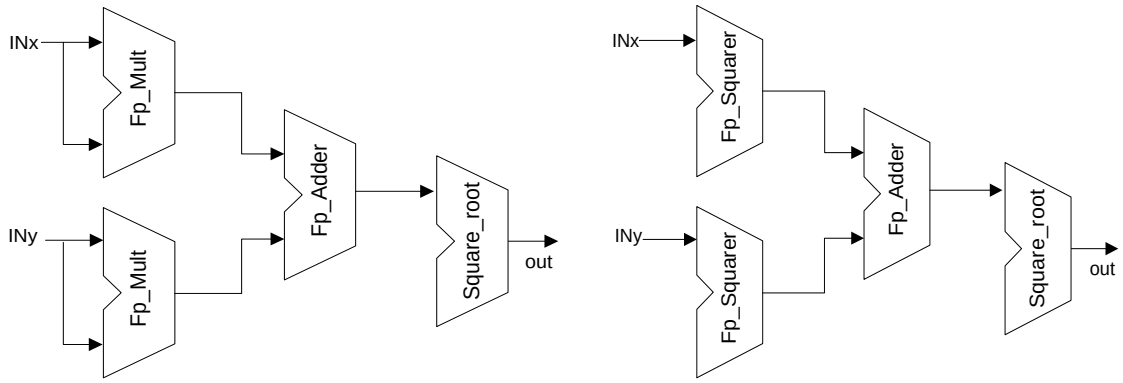
DISCRETE MAGNITUDE UNIT

The floating-point magnitude ($\sqrt{X^2 + Y^2}$) can be computed with discrete floating-point arithmetic operations. Figures 5.1 and 5.2 show serial and parallel versions of the floating-point discrete magnitude units with floating-point multipliers and squarers. For example, X^2 and Y^2 can be executed by a conventional floating-point squarer or multiplier. In a discrete parallel floating-point architecture, the squares/multiplications are performed in parallel. In the serial architecture, the two squares or multiplications are executed in serial and the result from the first square/multiplication is stored in a register. As a result, the discrete serial magnitude unit needs two cycles. After floating-point squares and multiplications, the floating-point addition is performed. Finally, the square-root of the sum of the squares is computed by a floating-point square-root unit.



a) with floating-point multiplier b) with floating-point squarer

Figure 5.1: Discrete Serial Floating-Point Magnitude Unit



a) with floating-point multiplier b) with floating-point squarer

Figure 5.2: Discrete Parallel Floating-Point Magnitude Unit

Floating-point Square-Root Design

A floating-point square-root unit is implemented for the discrete floating-point magnitude unit. Figure 5.3 shows a block diagram of a floating-point square-root unit in the discrete magnitude unit. In the floating-point square-root unit, the exponent is biased (+127) and divided by two before exponent adjustment. For the division by two, the biased exponent should be an even number. If not, the significand is shifted right by one and the biased exponent is adjusted to (exponent+1). These steps are called exponent ceiling [8]. The pre-normalized significand from the exponent ceiling operation is square-rooted by a single-precision signficand square-root unit. After the significand square-root computation, the range of the significand becomes$(\sqrt{1/2}, \sqrt{2})$, so that the square-root result needs to be normalized. The round unit needs a round bit. The round bit is obtained by an additional square-root chain [34]. Figure 5.4 shows the implementation of the floating-point square-root unit.

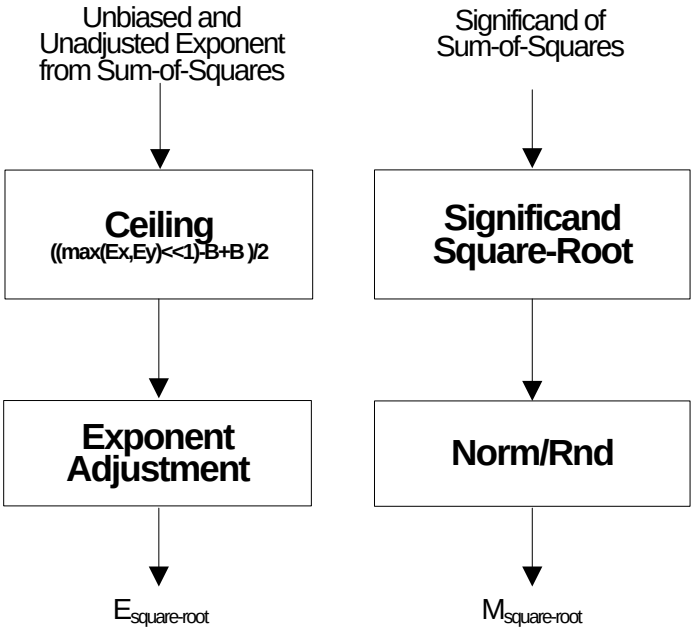


Figure 5.3: Floating-Point Square-Root Unit in the Discrete Magnitude Unit

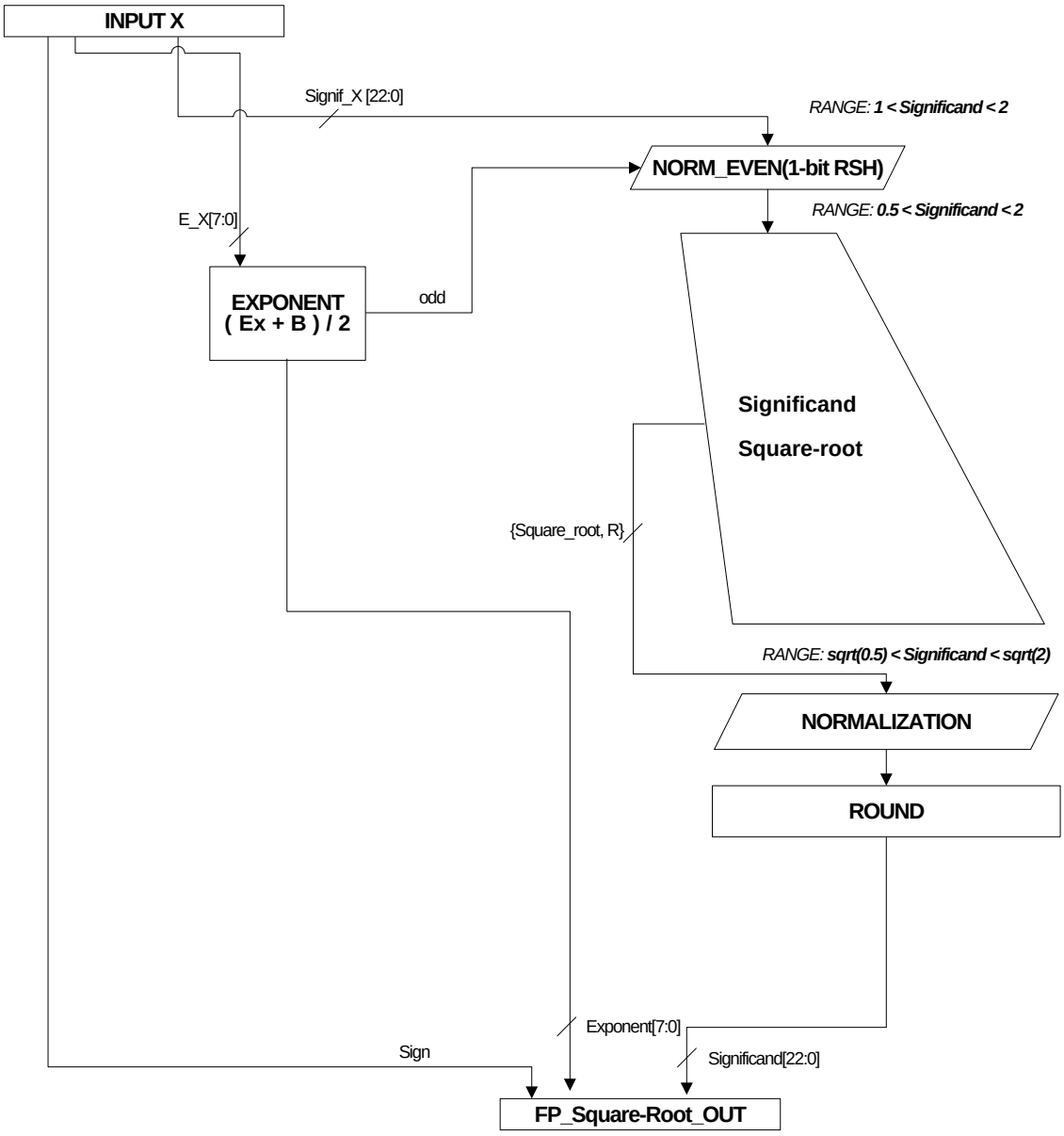


Figure 5.4: Implementation of the Floating-Point Square-Root Unit

Significand Square-Root

For the significand square-root computation, a binary digit recurrence non-restoring square-root unit is used. The advantages of the non-restoring algorithm are low power, small area, and easy rounding compared to the reciprocal square-root algorithm [35]. The digit recurrence algorithm can be divided into two categories: restoring and non-restoring methods. Figures 5.5 and 5.6 show the square-root computations using the restoring and non-restoring algorithm [34]. The figures use the following notations:

z (radicand) : $z_1 z_0 . z_{-1} \ldots\ldots z_{-\ell}$ $(1 \leqq z < 4)$

q (square-root) : $1 . q_{-1} q_{-2} q_{-3} \ldots\ldots q_{-\ell}$ $(1 \leqq q < 2)$

s (remainder) : $s_1 s_0 . s_{-1} \ldots\ldots s_{-\ell}$ $(0 \leqq q < 4)$

In the restoring square-root computation, when the remainder is a negative number, the root digit $q^{(j)}$ is zero and the partial remainder is restored. When $q_{-7} = 1$, the rounding-up is requested in Round to Nearest/Even (RNE) and Round-to-Plus-Infinity. For the non-restoring square-root computation, the root digit $q^{(j)}$ is -1 and $2q^{(j-1)} - 2^{-j}$ is added to the partial remainder for the next partial remainder when the remainder is a negative number.

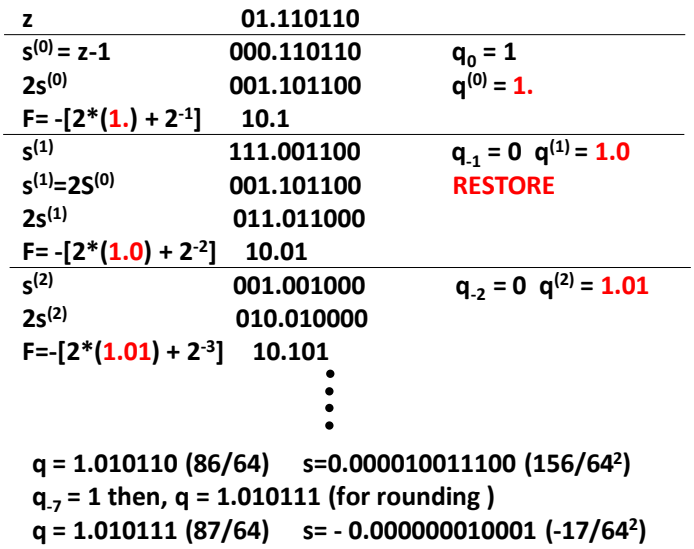


Figure 5.5: Square-Root Computation Using the Restoring Algorithm (after [34])

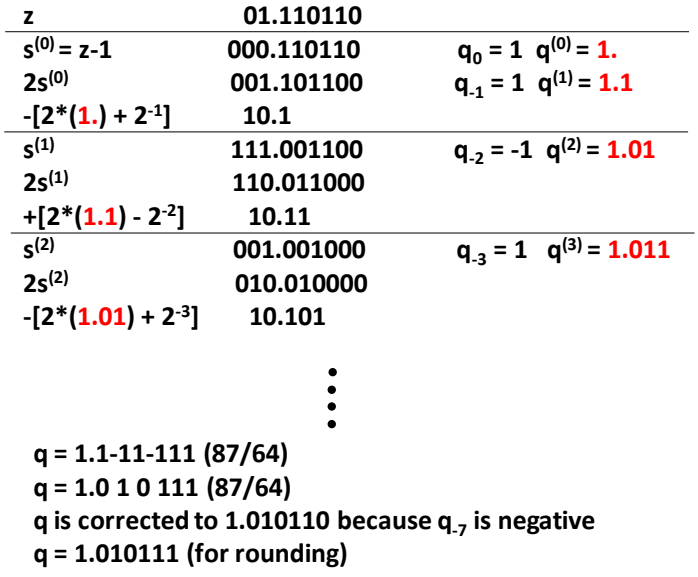


Figure 5.6: Square-Root Computation Using the Non-Restoring Algorithm (after [34])

The non-restoring algorithm is faster and requires less hardware than the restoring algorithm because the restoring algorithm has an additional step to restore the remainder. Due to the additional step, the restoring method is no longer used [36]. The non-restoring square-root algorithm has been used for several general purpose microprocessors and graphic processors [37]. Figure 5.7 represents implementation of the sequential non-restoring square-root unit [38].

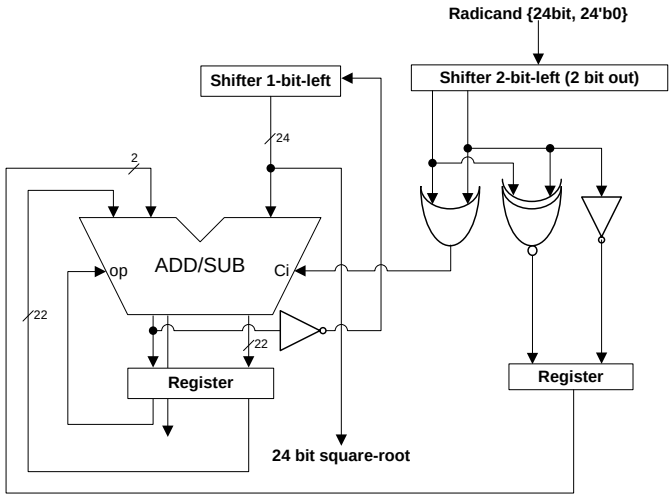


Figure 5.7: Sequential Non-restoring Square-Root Unit (after [38])

The sequential non-restoring square-root unit requires 24 clock phases for a single-precision floating-point square-root unit. However, it is difficult to adjust pipelined structures with fixed clock phases and internal registers of the sequential non-restoring unit. Therefore, a non-restoring radix-2 parallel array square-root unit (PASQRT) is applied in this dissertation [38,39]. Figure 5.8 illustrates a part of the block diagram of PASQRT. The structure of PASQRT is derived from the dot notation of a binary square-root algorithm. PASQRT can be used for combinational or pipelined ASIC structures.

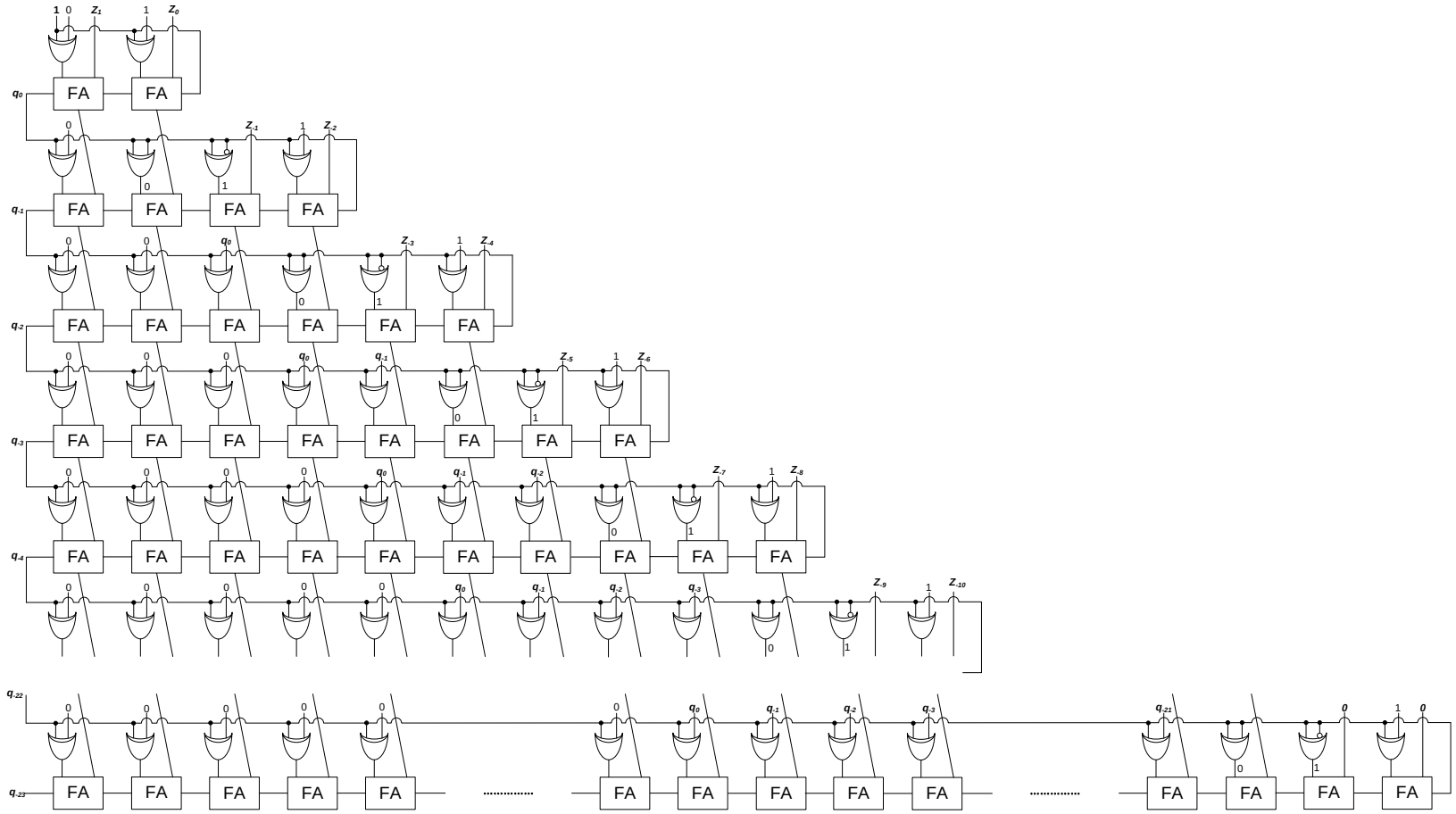


Figure 5.8: Primitive PASQRT Unit (after [34])

PROPOSED FUSED MAGNITUDE ARCHITECTURE

A new fused floating-point magnitude unit (fused MAG) is proposed for lower power consumption, area and latency. Figure 5.9 shows the block-diagram of the floating-point fused magnitude unit. The fused magnitude unit computes two 24 bit significand squares in parallel. The carry and sum words from the two significand squarers are swapped based on the exponent comparison. The swap unit is used to enable one-directional alignments. To reduce power consumption by the alignments, CSA, and significand adder, the Carry_net (carry generator) and Sticky unit are used for the bottom 21 bits as described in Chapter 4. When the exponents are unequal, the carry from the bottom bits is processed for sticky bit. The carry bits are added in a 4:2 CSA. After the alignment, the carry words and sum words from the two significand squarers are added by a CSA and a significand adder. The significand adder does not need the functions for subtraction such as Leading Zero Anticipation (LZA), two's complementation, and left shifting for normalization because the magnitude computation requires addition only. The result of the addition is pre-normalized, square-rooted and rounded. The pre-normalization is not for the exponent ceiling. The exponent unit of the fused magnitude unit does not have the exponent ceiling step. A compound add/round unit is suggested for the fused magnitude unit.

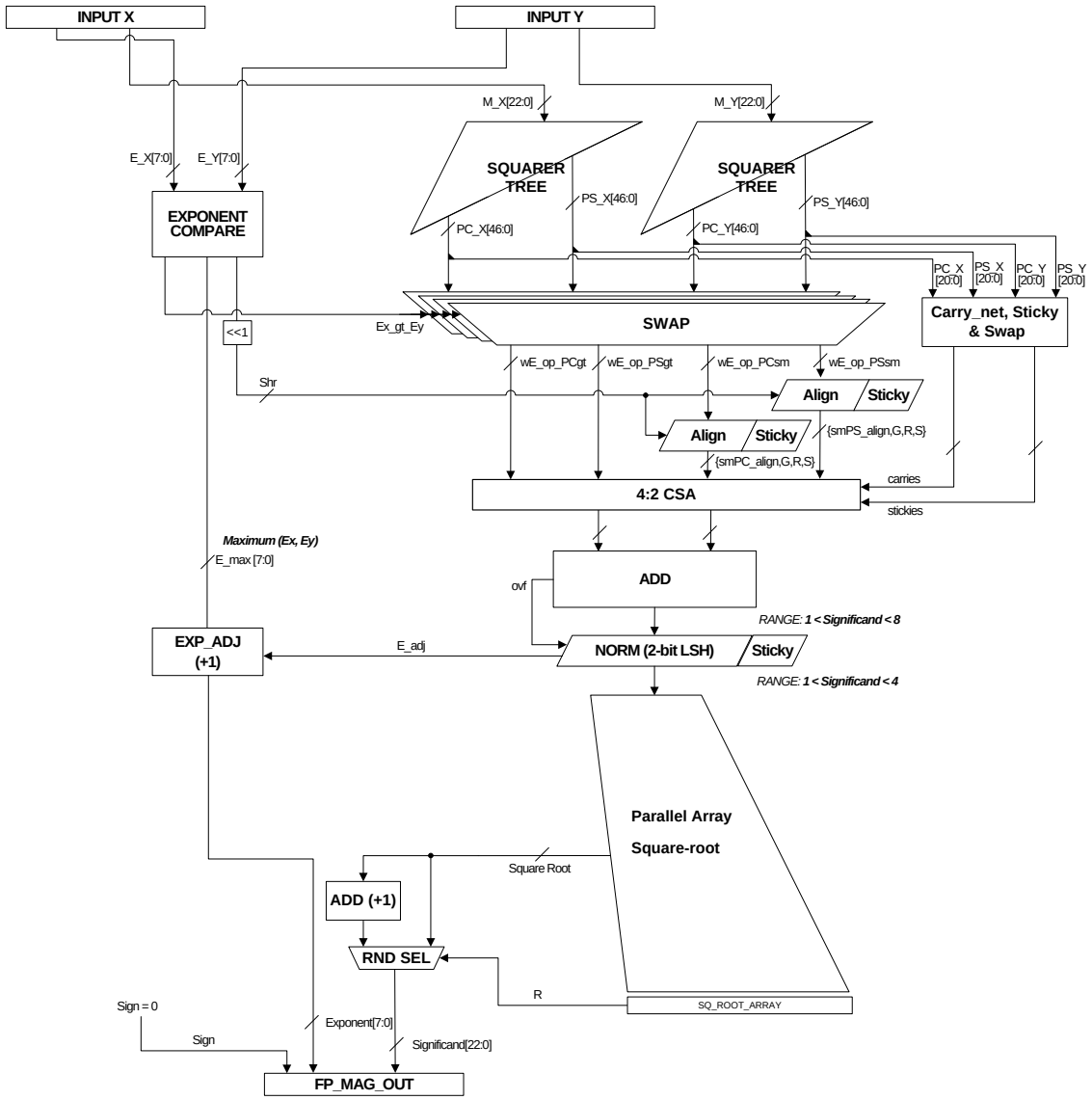


Figure 5.9: Proposed Floating-Point Fused Magnitude Unit

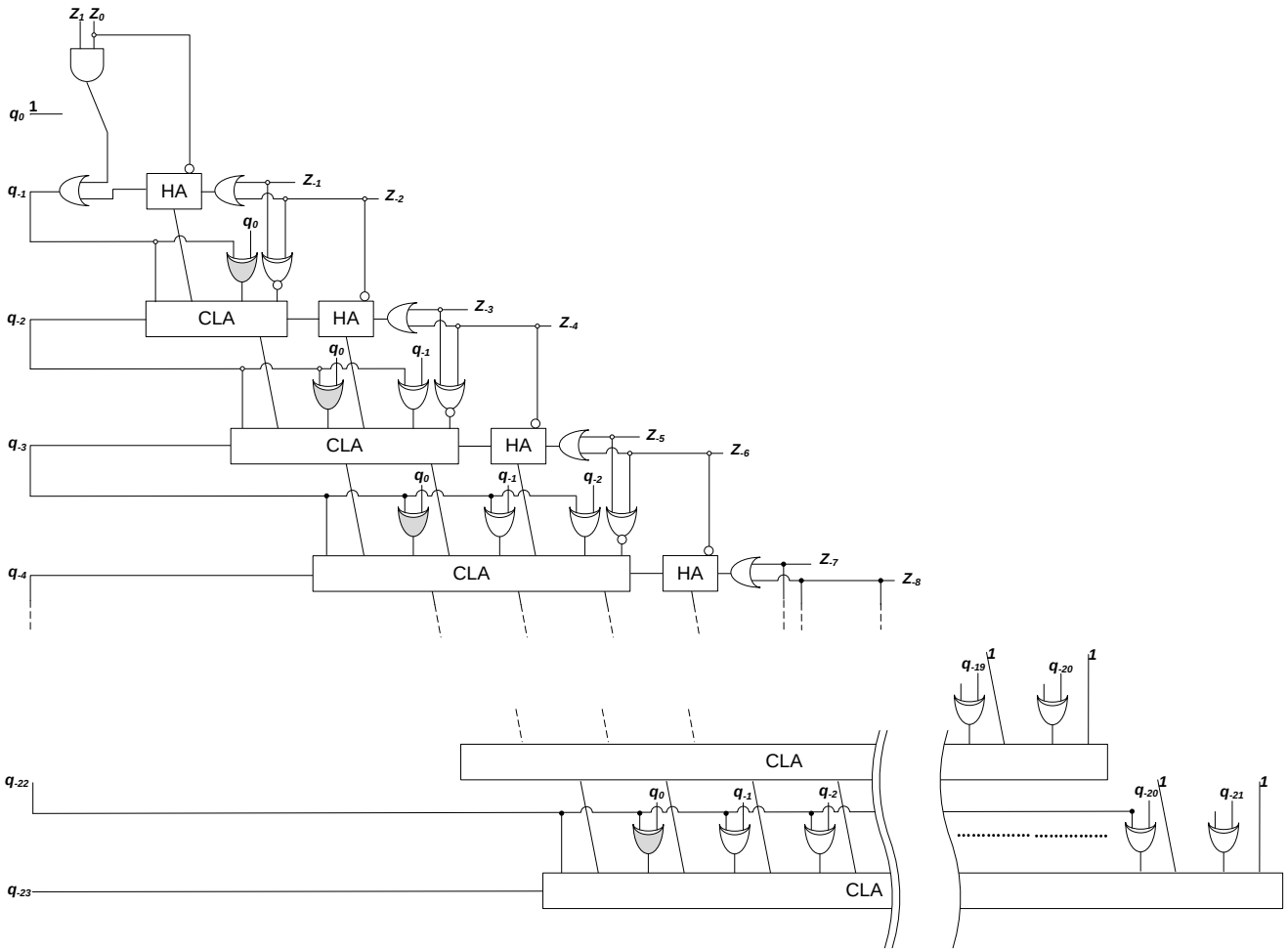


Figure 5.10: Significand Square-Root Unit

The significand square-root unit (PASQRT) of the fused magnitude unit is optimized and simplified for better performance. Figure 5.10 illustrates the block-diagram of the optimized significand square-root block. The input to the significand square-root block consists of two integer bits, 23 fractional bits, and Guard, Round, and Sticky bits. The PASQRT of the magnitude unit has a 3 bit larger input-width than that of a conventional floating-point square-root unit. The increased input width does not seriously degrade the area, power consumption, and latency. The increased input width only requires two half-adders and a few extra gates more, which are not located in the critical path. The MSB of the square-root result is always '1' because the input range of the significand square-root unit is (1, 4). As a result, the gray XOR gates of Figure 5.10 can be implemented as inverters. The sub-modules (i.e., pre-normalization, exponent compare, and compound add/round structure) are described in the following sections.

Table 5.1: Exponent Process in the Discrete Magnitude Unit

$(\sqrt{X^2 + Y^2})$		
Step1	$exp(X^2) = exp(X) \times 2 - Bias + E_{X^2_{adj}}$ $exp(Y^2) = exp(Y) \times 2 - Bias + E_{Y^2_{adj}}$	*for* X^2 *for* Y^2
Step2	$exp(X^2+Y^2) = MAX(X^2, Y^2) + E_{X^2+Y^2_{adj}}$	*for* X^2+Y^2
Step3	$exp(\sqrt{X^2 + Y^2}) = \{ exp(X^2+Y^2) + Bias \} / 2$	*for* $\sqrt{X^2 + Y^2}$

Pre-normalization

After the significand addition, the range of significand is (1, 8) (i.e., a three bit integer). With this range, the significand square-root unit generates a two bit integer significand (the range of the square-root result is (1, $\sqrt{8}$)). One more step of the square-root chain for MSB of the integer is required. To avoid the extra square-root step, pre-normalization is employed. If the significand result from the addition has three integer bits, the significand result is shifted right by 2 bits and the exponent is adjusted by +1.

Exponent Unit

The floating-point fused magnitude unit has a more enhanced exponent process than that of a discrete magnitude unit. Table 5.1 shows the conventional exponent processing steps. For the squares of X and Y, the exponents of the input X and Y are multiplied by 2 and biased by -127. The multiplied and biased exponents are added to exponent adjustment values from the normalization and round unit. In step 2, the larger exponent value of $exp(X^2)$ and $exp(Y^2)$ is selected for the valid exponent value and adjusted based on the normalization and rounding.

For step 3, the exponent from the addition is biased by +127 and divided by 2 (an exponent ceiling process is required). In the fused floating-point magnitude unit, these steps are merged and simplified. The eliminated steps reduce latency because the exponent adjustment steps from the round and exception process and the exponent ceiling process can be located in the critical path. Steps 1 and 2 can be merged:

$$exp(\ X^2+Y^2\ ) = MAX(\ exp(X), exp(Y)\ ) \times 2 - Bias + E_{adj} \quad (5.1)$$

For the merged exponent process, just one adjustment step is used. The combined exponent process can be merged again with step 3. The merged exponent computation is:

$$exp(\ \sqrt{X^2 + Y^2}\ ) = \{MAX(\ exp(X), exp(Y)\ ) << 1 - Bias + Bias\} >> 1 + E_{adj} \quad (5.2)$$

$$= MAX(\ exp(X), exp(Y)\ ) + E_{adj} \quad (5.3)$$

Multiplication and division by 2 can be replaced with 1-bit shifts. The two positive and negative bias values (+127, -127) cancel-out, so that overflow and underflow due to the bias processes does not need to be considered. The left-shift (multiplication by 2) and the right-shift (division by 2) also cancel-out. The simplified exponent computation is shown in Equation (5.3). The adjustment for Equation (5.3) comes from the pre-normalization.

Compound Add-Round

The compound addition/round step is suggested for improved performance. The additional square-root chain generates a round bit that used for the round process. For the round step, the round unit waits for the value of the round bit. The round unit computes rounded and unrounded results in parallel with the round bit. Then the round unit selects between the rounded and unrounded results after the value of the round bit is computed.

Pipeline Models

For the pipeline models (a 6 cycle pipeline for the fused and discrete parallel units and a 7 cycle pipeline for the serial units), the timing of the non-pipelined model is analyzed first. From the timing analysis, the first pipeline registers are inserted after the pre-normalization unit. The second, third, fourth, and fifth pipeline registers are inserted after the 7^{th}, 12^{th}, 16^{th}, and 20^{th} steps in the square-root chain. The last pipeline register is at the end of the fused magnitude unit.

IMPLEMENTATION & SIMULATION RESULTS

The non-pipelined and pipelined floating-point fused magnitude unit and conventional discrete units are implemented and synthesized with the 45-nm technology standard library. The non-pipelined designs were synthesized for the best performance by Synopsys Design Compiler (with ultra-compile no_autoungroup option) and compared under the same conditions of voltage, temperature and frequency with Synopsys PrimeTime-PX. For the pipelined models, the designs were synthesized for 300MHz performance. The power consumption was estimated under the same clock frequency (50MHz for the non-pipelined units and 300MHz for pipelined units) with the extracted RC parameters from the lay-out. Figure 5.11 shows the lay-out (110um by 110um) of the non-pipelined fused floating-point magnitude unit (the core density of the lay-out without physical cells is 77%). For the 6 cycle pipelined fused magnitude unit, the size of lay-out is 115um by 115um and the core density without physical cells is 73%.

A floating point addition-only-adder (half of a standard floating-point adder) is implemented for the discrete floating-point magnitude units with floating-point squarers to examine how much the fused magnitude architecture affects latency, power consumption, and area. The half floating-point adder does not have sub-modules for subtraction functions such as the sign-decision unit, 2's complement unit, normalization for left-shift and LZA.

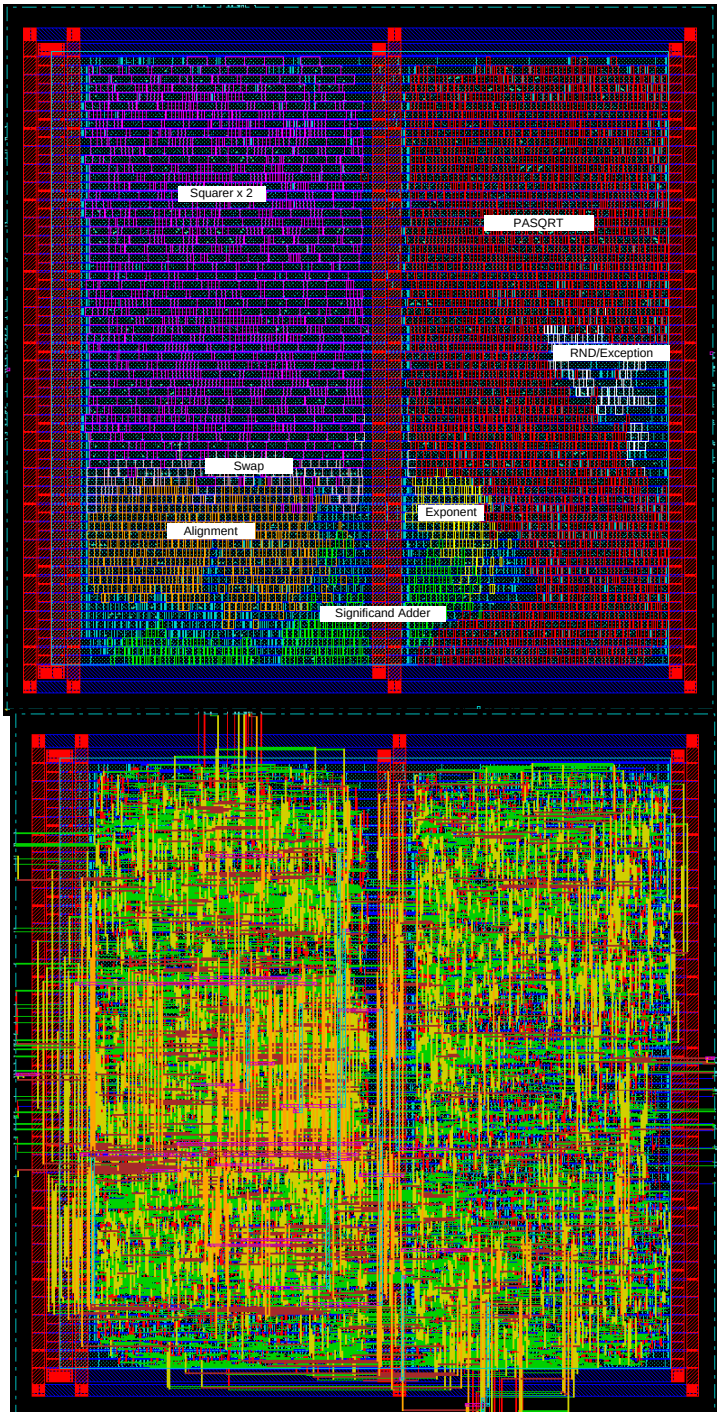


Figure 5.11: Layout of the Proposed Fused Magnitude Unit

Six types of discrete magnitude units were implemented and compared with the proposed fused magnitude unit. The results are shown in Table 5.2. To fully understand the fused magnitude unit, the non-pipelined units are compared for the cell area and latency.

Table 5.2 shows the cell area, latency for the non-pipelined magnitude units, and power consumption for both non-pipelined and pipelined magnitude units. Figure 5.12 shows the relative ratios of the power consumption, latency, and area relative to the fused magnitude unit. The proposed fused magnitude unit has less power consumption, area, and latency compared with the discrete parallel models.

Table 5.2: Comparison of Floating-Point Magnitude Units

Unit Type		**Standard Cell Area (μm^2)**	**Latency (ns)**	**Total Power (mW)**			
		Non-pipelined				Pipelined	
Discrete Serial	with FP Multiplier	9,317 (129%)	16.75 (154%)	2 cycles	1.15 (139%)	7 cycles	4.63 (144%)
	with FP Squarer / full FP adder	7,655 (106%)	16.03 (147%)		0.93 (112%)		3.59 (111%)
	with FP Squarer / half FP adder	6,418 (89%)	14.76 (136%)		0.81 (98%)		3.17 (98%)
Discrete Parallel	with FP Multiplier	12,833 (178%)	14.73 (136%)	1cycle	1.49 (180%)	6 cycles	6.18 (192%)
	with FP Squarer / full FP adder	9,508 (132%)	14.36 (132%)		1.13 (136%)		4.37 (136%)
	with FP Squarer / half FP adder	8,272 (115%)	13.09 (120%)		1.01 (122%)		3.95 (123%)
Fused Magnitude		7,216 (100%)	10.87 (100%)		0.83 (100%)		3.22 (100%)

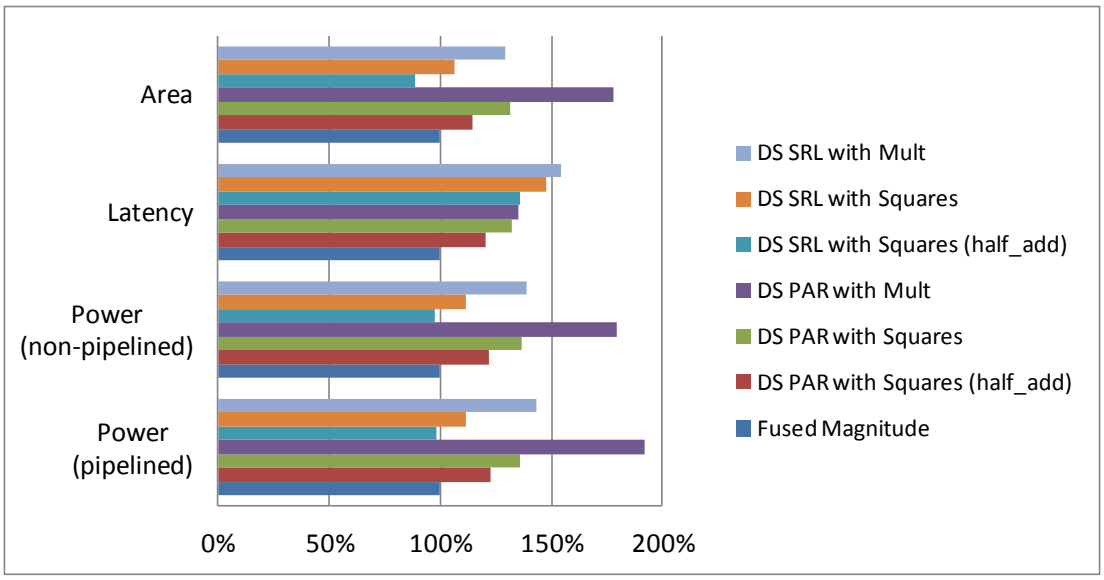


Figure 5.12: Relative Ratio of Performance Figures

Compared with the discrete parallel magnitude unit with floating-point squarers and half FP adder, the fused magnitude unit has 17.8% less power consumption in the non-pipelined model, 18.5% less power consumption in the pipelined model, 17.0% less latency, and 12.8% less area. The improvement is because the fused magnitude unit eliminates the round/norm/exception processes for floating-point squarers and addition and has enhanced exponent, compound add/round and normalization processes. Compared with the discrete parallel magnitude unit with floating-point squarers and full FP adder, the area, power consumption (non-pipelined and pipelined), and latency of the fused magnitude unit are 24.1%, (26.7%, 26.3%), and 24.3% less.

CHAPTER 6: CONCLUSION

This dissertation proposes new floating-point fused arithmetic designs for low-power dual-path add-subtract, sum-of-squares, and magnitude computation. The proposed fused arithmetic units are designed and implemented according to the industrial ASIC design flow to prove that the proposed architecture satisfies the goals of my dissertation. Synopsys and Cadence CAD tools and a 45 nm process library were used to implement the proposed designs. Python and MATLAB were used for verification and precision calculation.

In Chapter 3, the low power dual-path floating-point fused add-subtract unit was introduced. In contrast to the high-speed dual-path fused far/close path design, the far/close path of the low-power dual-path fused add-subtract unit does not contain the adders and subtractors. Due to the simple far/close path unit and shared adder and subtractors, the power consumption is reduced with the cost of increased latency. In addition, the proposed fused add-subtract unit has enhanced power-consumption, area, and latency compared to the parallel discrete add-subtract units due to the fused architecture. In Chapter 4, various floating-point fused two-term sum-of-squares architectures were introduced. Compared to the fused sum-of-squares unit with post-alignment, the fused sum-of-squares unit with pre-alignment has less latency and similar power consumption with a slight cost of increased area. The fused sum-of-squares unit with a compound add/rnd/norm unit has less latency with a moderate cost of increased power consumption and area. Compared to the fused sum-of-squares unit with full width carry-sum processing, the fused sum-of-squares unit with partial width carry-sum

processing has enhanced latency, area, and power consumption without losing precision. Therefore, among the various fused units, the fused sum-of-squares unit with compound add/rnd/norm, partial width carry-sum processing, and post-alignment unit has been proposed in this dissertation. However, the fused sum-of-squares units that have the combination of different options such as pre-alignment, different carry-sum width, enhanced rounding system, and square computation function can still provide more options for low-power and high-speed floating-point DSP applications. In Chapter 5, a new type of fused architecture for magnitude computation ($\sqrt{X^2 + Y^2}$) was proposed. The proposed fused magnitude unit has enhanced exponent processes and normalization. In addition, the pipelined magnitude unit was proposed. The fused floating-point arithmetic units proposed in this dissertation provide attractive design options for low-power and high-speed graphics, DSP and application-specific processing.

Portions of this dissertation have been published in the Asilomar Conference on Signals, Systems and Computers [40], the IEEE International Application-Specific Systems, Architectures and Processors [41], and the IEEE International Midwest Symposium on Circuits and Systems [42].

The proposed fused units require extra hardware resources. To reduce the extra hardware, a bridge architecture can be a solution. The bridge architecture performs fused operations with a specialized unit and existing basic floating-point arithmetic units such as floating-point adder, multiplier, and square-root unit. The bridge architecture for fused multiply-add unit has been proposed [43]. However, the bridge architectures for fused dot-product unit, sum-of-squares, and magnitude units have not been explored. Therefore, further research in the bridge architecture for fused arithmetic units is my future goal.

KEY CONTRIBUTIONS

This dissertation introduces improved floating-point fused architectures that have significant advantages compared to prior fused and discrete units. The new designs provide attractive options for low-power and high-speed floating-point applications. The improved fused architectures can be used for scientific and graphic applications. For example, the low-power fused dual-path add-subtract unit can be applied to a FFT unit with reduced power consumption and latency. The fused sum-of-squares unit can be used for Euclidian branching, pattern recognition, comparison of the magnitude of complex numbers, vector normalization, and digital filtering. The fused magnitude unit is used to compute the magnitude of complex numbers and vectors for radar signal processing and for conversion from rectangular to polar coordinates.

REFERENCES

[1] E. Hokenek, R. Montoye and P. W. Cook, "Second-Generation RISC Floating Point with Multiply-Add Fused," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 1207-1213, 1990.

[2] R. K. Montoye, E. Hokenek and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *IBM Journal of Research & Development*, vol. 34, pp. 59-70, 1990.

[3] H. H. Saleh and E. E. Swartzlander, Jr., "A Floating-point Fused Add-Subtract Unit," *The Fifty-First Midwest Symposium on Circuits and Systems, MWSCAS*, pp. 519-522, 2008.

[4] H. H. Saleh and E. E. Swartzlander, Jr., *Floating-Point Fused Add-Subtract Unit*, US Patent 8,161,090, 2012.

[5] J. Sohn and E. E. Swartzlander, Jr., "Improved Architectures for a Fused Floating-Point Add-Subtract Unit," *IEEE Transaction on Circuits and Systems-I*, vol 59, pp. 2285-2291, Oct. 2012.

[6] H. H. Saleh and E. E. Swartzlander, Jr., "A Floating-point Fused Dot-Product Unit," *IEEE International Conference on Computer Design, ICCD*, pp. 427-431, 2008.

[7] H. H. Saleh and E. E. Swartzlander, Jr., *Floating-Point Fused Dot-Product Unit*, US Patent 8166091, 2012.

[8] M. Ercegovac and T. Lang, *Digital Arithmetic*, San Francisco: Morgan-Kaufmann Publishers, 2006.

[9] J. Rupley, J. King, E. Quinnell, F. Galloway, K. Patton, P. Seidel, J. Dinh, H. Bui, and A. Bhowmik, "The Floating-Point Unit of the Jaguar x86 Core," *The 21st IEEE Symposium on Computer Arithmetic*, pp. 7-16, 2013.

[10] "SPARC," The SPARC Architecture Manual Version 8.

[11] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008.

[12] W. J. Cody, J. T. Coonen, D. M. Gay, K. Hanson, W. Kahan, R. Karpinski, J. Palmer, F. N. Ris, and D. Stevenson "A proposed radix-and-word-lengh-independent standard for floating-point arithmetic," *IEEE MICRO*, vol. 4, no. 4 pp. 86-100, 1984.

[13] S. Vassiliadis, D.S. Lemonm and M. Putrino, "S/370 sign-magnitude floating-point adder," *IEEE Journal of Solid-State Circuits*, vol. 24, pp. 1062-1070, 1989.

[14] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim, "Reduced latency IEEE floating-point standard adder architectures," *The 14th IEEE Symposium on Computer Arithmetic*, pp. 35-42, 1999.

[15] P.-M. Seidel, and G. Even, "On the design of fast IEEE floating-point adders," *The 15th IEEE Symposium on Computer Arithmetic*, pp. 184-194, 2001.

[16] E. E. Swartzlander, Jr., "*EE382V Floating-point Arithmetic and Design Course Notes,"* University of Texas at Austin, 2006.

[17] E. Quinnell, *Floating-Point Fused Multiply-Add Architectures*, Ph.D. Dissertation, Dept. Electrical and Computer Engineering, University of Texas at Austin, 2007.

[18] M. P. Farmwald, "*On the Design of High Performance Digital Arithmetic Units*," Ph.D. Dissertation, Dept. Comput. Sci., Stanford University, Stanford, CA, 1981.

[19] M. Uya, K. Kaneko, and J.Yasui, " A CMOS floating point multiplier," *IEEE Journal of Solid-State Circuits*, vol. 19, pp. 697-702, 1984.

[20] M. R. Santoro, G. Bewick, and M. A. Horowitz, " Rounding algorithms for IEEE multipliers," *The 9th IEEE Symposium on Computer Arithmetic*, pp. 176-183, 1989.

[21] "Python," *The Python Tutorial 3.3.2.*

[22] "Synopsys," VSC® /VCSi™ User Guide Y-2006.06-SP2.

[23] "Cadence," *SoC Encounter User Guide, Version 4.1.5.*

[24] "Synopsys," *Design Compiler User Guide, Version G-2012.06.*

[25] "Synopsys," *PrimeTime and PrimeTime PX User Guide, Version G-2012.06.*

[26] E. E. Swartzlander, Jr. and H. H. Saleh, "Fused floating-point arithmetic for DSP," *The Forty-Second Asilomar Conference on Signals, Systems and Computers, ACSSC*, pp. 767-771, 2008.

[27] H. H. M. Saleh, *Fused Floating-Point Arithmetic For DSP*, Ph.D. Dissertation, Dept. Electrical and Computer Engineering, University of Texas at Austin, 2009.

[29] M. J. Schulte, L. Marquette, S. Krithivasan, E.G. Walters, and J. Glossner, “Combined Multiplication and Sum-of-Squares Units,” *IEEE International Conference on Application-Specific System, Architectures and Processors, ASAP*, pp. 204-214, 2003.

[30] F. de Dinechin, C. Klein, and B. Pasca, “Generating high-performance custom floating-point pipelines,” *International Conference on Field Programmable Logic and Applications, FPA*, pp. 59-64, 2009.

[31] T. C. Chen, “A binary multiplication scheme based on squaring,” *IEEE Transactions on Computers*, vol. C-20, pp. 678-680, 1971.

[32] J. A. Lunsford, *Square Root of Sum of Squares Approximator*, US Patent 3,858,036, 1974.

[33] A. E. Filip, “A Baker's Dozen Magnitude Approximations and Their Detection Statistics,” *IEEE Transaction on Aerospace and Electronic Systems*, vol. AES-12, pp. 86-89, 1976.

[34] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, New York: Oxford, 2000.

[35] W. Liu and A. Nannarelli, "Power Efficient Division and Square Root Unit," *IEEE Transactions on Computers*, vol. 61, pp. 1059-1070, 2012.

[36] T. Sutikno, "An Efficient Implementation of the NonRestoring Square Root Algorithm in Gate Level," *International Journal of Computer Theory and Engineering*, vol. 3, pp. 46-52, 2011.

[37] E. Antelo, T. Lang, P. Montuschi, and A. Nannarelli, "Low latency digit-recurrence reciprocal and square-root reciprocal algorithm and architecture," *The 17th IEEE Symposium on Computer Arithmetic*, pp. 147-154, 2005.

[38] L. Yamin and C. Wanming, "Implementation of Single PrecisionFloating Point Square Root on FPGAs," *IEEE Symposium on FPGA for Custom Computing Machines*, pp. 226-232, 1997.

[39] L. Yamin and C. Wanming, "Parallel-array implementations of a non-restoring square root algorithm," *IEEE International Conference on Computer Design, ICCD*, pp. 690-695, 1997.

[40] J. Min, J. Sohn, and E. E. Swartzlander, Jr., "A Low-Power Dual-Path Floating-Point Fused Add-Subtract Unit," *The Forty-Sixth Asilomar Conference on Signals, System and Computers, ACSSC*, pp.998-2002, 2012.

[41] J. Min and E. E. Swartzlander, Jr., "Fused Floating-Point Two-term Sum-of-Squares Unit," *The Twenty-Forth IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP*, pp. 147-152, 2013.

[42] J. Min and E. E. Swartzlander, Jr., "Fused Floating-Point Magnitude Unit," *The Fifty-Sixth IEEE International Midwest Symposium on Circuits and Systems, MWSCAS*, 2013.

[43] E. Quinnell, E. E. Swartzlander, Jr., and C. Lemonds, "Bridge Floating-Point Fused Multiply-Add Design," *IEEE Transaction on VLSI Systems*, vol. 16, pp.1727-1731, 2008.

VITA

Jae Hong Min was born in Gwangmyeong, South Korea on December 26th, 1981. He is the son of Kyeongjin Min and Jungae Jung. Based on the dual degree program, he received his Bachelor degrees in Electrical Engineering from both Ajou University, in Suwon, South Korea and State University of New York, Stony Brook in Stony Brook, New York in 2008. He enrolled in the Electrical and Computer Engineering Ph.D. program at the University of Texas at Austin in the Fall 2008.

Email Address: jmin@utexas.edu

This dissertation was typed by Jae Hong Min.